GEOPHYSICAL ESTIMATION BY EXAMPLE

Jon F. Claerbout with Sergey Fomel

Directory

Last Revision Date: March 12, 2008

dedicated to the memory

of

Johannes “Jos” Claerbout

1974-1999

“What do we have to look forward to today?
There are a lot of things we have to look forward to

today.”

`http://sep.stanford.edu/sep/jon/family/jos/`

Table of Contents

- Autocorr - compute autocorrelation for helix filters
- Bin2 - nearest neighbor binning in 2-D
- Conv - convolve two helix filters
- Decon - Deconvolution (N-dimensional)
- Devector - create one output filter from two input filters
- Helderiv - Helix derivative filter in 2-D
- Helicon - Helix convolution and deconvolution (N-dimensional!)
- Helocut - Helix Lowcut filter in 2-D
- Hole - Punch ellipsoidal hole in 2-D data
- Igrad - take gradient on the first axis
- LPad - Pad and interleave traces
- LPef - Find PEF on aliased traces
- Lapfill2 - fill missing data by minimizing the Laplacian
- LoLPef - Find PEF on aliased traces (with patching)
- Lomiss - Missing data interpolation with a prescribed helix filter
- (in local patches)
- Lopef - Local Prediction-Error Filter (1-D, 2-D, and 3-D)
- Losignoi - Local signal and noise separation (N-dimensional)
- MSHelicon - Multi-scale Helix convolution (N-dimensional!)
- MSMiss - Multiscale missing data interpolation (N-dimensional)
- MSPef - Multi-scale PEF estimation
- Make - generate simple 2-D synthetics with crossing plane waves
- Minphase - create minimum-phase filters
- Miss - Missing data interpolation with a prescribed helix filter
- NHelicon - Non-stationary helix convolution and deconvolution
- NPef - Estimate

Preface

The difference between theory and practice is smaller in theory than it is in practice. –folklore

We make discoveries about reality by examining the discrepancy between theory and practice. There is a well-developed *theory* about the difference between theory and practice, and it is called "geophysical inverse theory". In this book we investigate the *practice* of the difference between theory and practice. As the folklore tells us, there is a big difference. There are already many books on the theory, and

often as not, they end in only one or a few applications in the author's specialty. In this book on practice, we examine data and results from many diverse applications. I have adopted the discipline of suppressing theoretical curiosities until I find data that requires it (except for a few concepts at chapter ends).

Books on geophysical inverse theory tend to address theoretical topics that are little used in practice. Foremost is probability theory. In practice, probabilities are neither observed nor derived from observations. For more than a handful of variables, it would not be practical to display joint probabilities, even if we had them. If you are data poor, you might turn to probabilities. If you are data rich, you have far too many more rewarding things to do. When you estimate a few values, you ask about their standard deviations. When you have an image making machine, you turn the knobs and make new images (and invent new knobs). Another theory not needed here is singular-value decomposition.

In writing a book on the "practice of the difference between theory and practice" there is no worry to be bogged down in the details of diverse specializations because the geophysical world has many interesting data sets that are easily analyzed with elementary physics and simple geometry. (My specialization, reflection seismic imaging, has a great many less easily explained applications too.) We find

here many applications that have a great deal in common with one another, and that commonality is not a part of common inverse theory. Many applications draw our attention to the importance of two weighting functions (one required for data space and the other for model space). Solutions depend strongly on these weighting functions (eigenvalues do too!). Where do these functions come from, from what rationale or estimation procedure? We'll see many examples here, and find that these functions are not merely weights but filters. Even deeper, they are generally a combination of weights and filters. We do some tricky bookkeeping and bootstrapping when we filter the multidimensional neighborhood of missing and/or suspicious data.

Are you aged 23? If so, this book is designed for you. Life has its discontinuities: when you enter school at age 5, when you leave university, when you marry, when you retire. The discontinuity at age 23, mid graduate school, is when the world loses interest in your potential to learn. Instead the world wants to know what you are accomplishing right now! This book is about how to make images. It is theory and programs that you can use right now.

This book is not devoid of theory and abstraction. Indeed it makes an important new contribution to the theory (and practice) of data analysis: multidimensional

autoregression via the helical coordinate system.

The biggest chore in the study of “the practice of the difference between theory and practice" is that we must look at algorithms. Some of them are short and sweet, but other important algorithms are complicated and ugly in any language. This book can be printed without the computer programs and their surrounding paragraphs, or you can read it without them. I suggest, however, you take a few moments to try to read each program. If you can *write* in any computer language, you should be able to *read* these programs well enough to grasp the concept of each, to understand what goes in and what should come out. I have chosen the computer language (more on this later) that I believe is best suited for our journey through the “elementary” examples in geophysical image estimation.

Besides the tutorial value of the programs, if you can read them, you will know exactly how the many interesting illustrations in this book were computed so you will be well equipped to move forward in your own direction.

THANKS

2006 is my fourteenth year of working on this book and much of it comes from earlier work and the experience of four previous books. In this book, as in my previous books, I owe a great deal to the many students at the Stanford Exploration Project. I would like to mention some with particularly notable contributions (in approximate historical order).

The concept of this book began along with the PhD thesis of Jeff Thorson. Before that, we imagers thought of our field as "an hoc collection of good ideas" instead of as "adjoints of forward problems". Bill Harlan understood most of the preconditioning issues long before I did. All of us have a longstanding debt to Rick Ottolini who built a cube movie program long before anyone else in the industry had such a blessing.

My first book was built with a typewriter and ancient technologies. In early days each illustration would be prepared without reusing packaged code. In assembling my second book I found I needed to develop common threads and code them only once and make this code systematic and if not idiot proof, then “idiot resistant”. My early attempts to introduce “`seplib`” were not widely welcomed until Stew Levin rebuilt everything making it much more robust. My second book was

typed in the troff text language. I am indebted to Kamal Al-Yahya who not only converted that book to LaTeX, but who wrote a general-purpose conversion program that became used internationally.

Early days were a total chaos of plot languages. I and all the others at SEP are deeply indebted to Joe Dellinger who starting from work of Dave Hale, produced our internal plot language "vplot" which gave us reproducibiliy and continuity over decades. Now, for example, our plots seamlessly may be directed to postscript (and PDF), Xwindow, or the web. My second book required that illustrations be literally taped onto the sheet containing the words. All of us benefitted immensely from the work of Steve Cole who converted Joe's vplot language to postscript which was automatically integrated with the text.

When I began my third book I was adapting liberally from earlier work. I began to realize the importance of being able to reproduce any earlier calculation and began building rules and file-naming conventions for "reproducible research". This would have been impossible were it not for Dave Nichols who introduced `cake`, a variant of the UNIX software building program `make`. Martin Karrenbach continued the construction of our invention of "reproducible research" and extended it to producing reproducible research reports on CD-ROM, an idea well ahead of

its time. Some projects were fantastic for their time but had the misfortune of not being widely adopted, ultimately becoming unsupportable. In this catagory was Dave and Martin's implementation `xtex`, a magnificent way of embedding reproducible research in an electronic textbook. When `cake` suffered the same fate as `xtex`, Matthias Schwab saved us from mainstream isolation by bringing our build procedures into the popular GNU world.

Coming to the present textbook I mention Bob Clapp. He made numerous contributions. When Fortran77 was replaced by Fortran90, he rewrote Ratfor. For many years I (and many of us) depended on Ratfor as our interface to Fortran and as a way of presenting uncluttered code. Bob rewrote Ratfor from scratch merging it with other SEP-specific software tools (Sat) making Ratfor90. Bob prepared the interval-velocity examples in this book. Bob also developed most of the "geostat" ideas and examples in this book. Morgan Brown introduced the texture examples that we find so charming. Paul Sava totally revised the book's presentation of least-squares solvers making them more palatable to students and making more honest our claim that in each case the results you see were produced by the code you see.

One name needs to be singled out. Sergey Fomel converted all the examples in this book from my original Fortran 77 to a much needed modern style of Fortran

90. After I discovered the helix idea and its wide-ranging utility, he adapted all the relevant examples in this book to use it. If you read Fomel's programs, you can learn effective application of that 1990's revolution in coding style known as "object orientation."

March 12, 2008

Overview

This book is about the estimation and construction of geophysical images. Geophysical images are used to visualize petroleum and mineral resource prospects, subsurface water, contaminent transport (environmental pollution), archeology, lost treasure, even graves.

Here we follow physical measurements from a wide variety of geophysical sounding devices to a geophysical image, a 1-, 2-, or 3-dimensional Cartesian mesh that is easily transformed to a graph, map image, or computer movie. A later more

human, application-specific stage (not addressed here) interprets and annotates the images; that stage places the "$\times$" where you will drill, dig, dive, or merely dream.

Image estimation is a subset of "geophysical inverse theory," itself a kind of "theory of how to find everything." In contrast to "everything," images have an organized structure (covariance) that makes their estimation more concrete and visual, and leads to the appealing results we find here.

Geophysical sounding data used in this book comes from acoustics, radar, and seismology. Sounders are operated along tracks on the earth surface (or tracks in the ocean, air, or space). A basic goal of data processing is an image that shows the earth itself, not an image of our data-acquisition tracks. We want to hide our data acquisition footprint. Increasingly, geophysicists are being asked to measure changes in the earth by comparing old surveys to new ones. Then we are involved with both the old survey tracks and new ones, as well as technological changes between old sounders and new ones.

To enable this book to move rapidly along from one application to another, we avoid applications where the transform from model to data is mathematically complicated, but we include the central techniques of constructing the adjoint of any such complicated transformation. By setting aside application-specific complica-

tions, we soon uncover and deal with universal difficulties such as: (1) irregular geometry of recording, (2) locations where no recording took place and, (3) locations where crossing tracks made inconsistant measurements because of noise. Noise itself comes in three flavors: (1) drift (zero to low frequency), (2) white or steady and stationary broad band, and (3) bursty, i.e., large and erratic.

Missing data and inconsistant data are two humble, though universal problems. Because they are universal problems, science and engineering have produced a cornucopia of ideas ranging from mathematics (Hilbert adjoint) to statistics (inverse covariance) to conceptual (stationary, scale-invariant) to numerical analysis (conjugate direction, preconditioner) to computer science (object oriented) to simple common sense. Our guide through this maze of opportunities and digressions is the test of what works on real data, what will make a better image. My logic for organizing the book is simply this: Easy results first. Harder results later. Undemonstrated ideas last or not at all, and latter parts of chapters can be skimmed.

Examples here are mostly nonseismological although my closest colleagues and I mostly make images from seismological data. The construction of 3-D subsurface landform images from seismological data is an aggressive industry, a complex and competitive place where it is not easy to build yourself a niche. I wrote this

book because I found that beginning researchers were often caught between high expectations and concrete realities. They invent a new process to build a novel image but they have many frustrations: (1) lack of computer power, (2) data-acquisition limitations (gaps, tracks, noises), or (3) they see chaotic noise and have difficulty discerning whether the noise represents chaos in the earth, chaos in the data acquisition, chaos in the numerical analysis, or unrealistic expectations.

People need more practice with easier problems like the ones found in this book, which are mostly simple 2-D landforms derived from 2-D data. Such concrete estimation problems are solved quickly, and their visual results provide experience in recognizing weaknesses, reformulating, and moving forward again. Many small steps reach a mountain top.

Scaling up to big problems

Although most the examples in this book are presented as toys, where results are obtained in a few minutes on a home computer, we have serious industrial-scale jobs always in the backs of our minds. This forces us to avoid representing operators as matrices. Instead we represent operators as a pair of subroutines, one to apply the

operator and one to apply the adjoint (transpose matrix). (This will be more clear when you reach the middle of chapter 2.)

By taking a function-pair approach to operators instead of a matrix approach, this book becomes a guide to practical work on realistic-sized data sets. By realistic, I mean as large and larger than those here; i.e., data ranging over two or more dimensions, and the data space and model space sizes being larger than about 10^5 elements, about a 300×300 image. Even for these, the world's biggest computer would be required to hold in random access memory the $10^5 \times 10^5$ matrix linking data and image. Mathematica, Matlab, kriging, etc, are nice tools but[1] it was no surprise when a curious student tried to apply one to an example from this book and discovered that he needed to abandon 99.6% of the data to make it work. Matrix methods are limited not only by the size of the matrices but also by the fact that the cost to multiply or invert is proportional to the third power of the size. For simple experimental work, this limits the matrix approach to data and images of about 4000 elements, a low-resolution 64×64 image.

---

[1] I do not mean to imply that these tools cannot be used in the function-pair style of this book, only that beginners tend to use a matrix approach.

Chapter 1

Basic operators and adjoints

A great many of the calculations we do in science and engineering are really matrix multiplication in disguise. The first goal of this chapter is to unmask the disguise by showing many examples. Second, we see how the **adjoint** operator (matrix

transpose) back projects information from data to the underlying model.

Geophysical modeling calculations generally use linear operators that predict data from models. Our usual task is to find the inverse of these calculations; i.e., to find models (or make images) from the data. Logically, the adjoint is the first step and a part of all subsequent steps in this **inversion** process. Surprisingly, in practice the adjoint sometimes does a better job than the inverse! This is because the adjoint operator tolerates imperfections in the data and does not demand that the data provide full information.

Using the methods of this chapter, you will find that once you grasp the relationship between operators in general and their adjoints, you can obtain the adjoint just as soon as you have learned how to code the modeling operator.

If you will permit me a poet's license with words, I will offer you the following table of **operator**s and their **adjoint**s:

matrix multiply	conjugate-transpose matrix multiply
convolve	crosscorrelate
truncate	zero pad
replicate, scatter, spray	sum or stack
spray into neighborhoods	sum within bins

derivative (slope)	negative derivative
causal integration	anticausal integration
add functions	do integrals
assignment statements	added terms
plane-wave superposition	slant stack / beam form
superpose curves	sum along a curve
stretch	squeeze
scalar field gradient	negative of vector field divergence
upward continue	downward continue
diffraction modeling	imaging by migration
hyperbola modeling	CDP stacking
ray tracing	**tomography**

The left column above is often called "**modeling**," and the adjoint operators on the right are often used in "data **processing**."

When the adjoint operator is *not* an adequate approximation to the inverse, then you apply the techniques of fitting and optimization explained in Chapter 2. These techniques require iterative use of the modeling operator and its adjoint.

The adjoint operator is sometimes called the "**back projection**" operator because information propagated in one direction (earth to data) is projected backward (data to earth model). Using complex-valued operators, the transpose and complex conjugate go together; and in **Fourier analysis**, taking the complex conjugate of $\exp(i\omega t)$ reverses the sense of time. With more poetic license, I say that adjoint operators *undo* the time and phase shifts of modeling operators. The inverse operator does this too, but it also divides out the color. For example, when linear interpolation is done, then high frequencies are smoothed out, so inverse interpolation must restore them. You can imagine the possibilities for noise amplification. That is why adjoints are safer than inverses. But nature determines in each application what is the best operator to use, and whether to stop after the adjoint, to go the whole way to the inverse, or to stop partway.

The operators and adjoints above transform vectors to other vectors. They also transform data planes to model planes, volumes, etc. A mathematical operator transforms an "abstract vector" which might be packed full of volumes of information like television signals (time series) can pack together a movie, a sequence of frames. We can always think of the operator as being a matrix but the matrix can be truly huge (and nearly empty). When the vectors transformed by the matrices are large

like geophysical data set sizes then the matrix sizes are "large squared," far too big for computers. Thus although we can always think of an operator as a matrix, in practice, we handle an operator differently. Each practical application requires the practitioner to prepare two computer programs. One performs the matrix multiply $\mathbf{y} = \mathbf{A}\mathbf{x}$ and another multiplys by the transpose $\tilde{\mathbf{x}} = \mathbf{A}'\mathbf{y}$ (without ever having the matrix itself in memory). It is always easy to transpose a matrix. It is less easy to take a computer program that does $\mathbf{y} = \mathbf{A}\mathbf{x}$ and convert it to another to do $\tilde{\mathbf{x}} = \mathbf{A}'\mathbf{y}$. In this chapter are many examples of increasing complexity. At the end of the chapter we will see a test for any program pair to see whether the operators $\mathbf{A}$ and $\mathbf{A}'$ are mutually adjoint as they should be. Doing the job correctly (coding adjoints without making approximations) will reward us later when we tackle model and image estimation problems.

1.0.1. Programming linear operators

The operation $y_i = \sum_j b_{ij} x_j$ is the multiplication of a matrix $\mathbf{B}$ by a vector $\mathbf{x}$. The adjoint operation is $\tilde{x}_j = \sum_i b_{ij} y_i$. The operation adjoint to multiplication by a matrix is multiplication by the transposed matrix (unless the matrix has complex

elements, in which case we need the complex-conjugated transpose). The following **pseudocode** does matrix multiplication $\mathbf{y} = \mathbf{B}\mathbf{x}$ and multiplication by the transpose $\tilde{\mathbf{x}} = \mathbf{B}'\mathbf{y}$:

```
if adjoint
        then erase x
if operator itself
        then erase y
do iy = 1, ny {
do ix = 1, nx {
        if adjoint
                x(ix) = x(ix) + b(iy,ix) × y(iy)
        if operator itself
                y(iy) = y(iy) + b(iy,ix) × x(ix)
        }}
```

Notice that the "bottom line" in the program is that *x* and *y* are simply interchanged. The above example is a prototype of many to follow, so observe carefully the similarities and differences between the adjoint and the operator itself.

```
module matmult {   # matrix multiply and its adjoint
real, dimension (:,:), pointer :: bb
#%  _init( bb)
#%  _lop(  x, y)
integer ix, iy
do ix= 1, size(x) {
do iy= 1, size(y) {
        if( adj)
                x(ix) = x(ix) + bb(iy,ix) * y(iy)
        else
                y(iy) = y(iy) + bb(iy,ix) * x(ix)
        }}
}
```

Back

Next we restate the matrix-multiply pseudo code in real code, in a language called **Loptran**[1], a language designed for exposition and research in model fitting and optimization in physical sciences. The module `matmult` for matrix multiply and its adjoint exhibits the style that we will use repeatedly. At last count there were 53 such routines (operator with adjoint) in this book alone. matmult Notice that the module `matmult` does not explicitly erase its output before it begins, as does the psuedo code. That is because Loptran will always erase for you the space required for the operator's output. Loptran also defines a logical variable `adj` for you to distinguish your computation of the adjoint `x=x+B'*y` from the forward operation `y=y+B*x`. In computerese, the two lines beginning #% are macro expansions that take compact bits of information which expand into the verbose boilerplate that Fortran requires. Loptran is Fortran with these macro expansions. You can always see how they expand by looking at `http://sepwww.stanford.edu/sep/prof/`.

What is new in Fortran 90, and will be a big help to us, is that instead of a subroutine with a single entry, we now have a module with two entries, one named

---

[1] The programming language, Loptran, is based on a dialect of Fortran called Ratfor. For more details, see Appendix A.

`_init` for the physical scientist who defines the physical problem by defining the matrix, and another named `_lop` for the least-squares problem solver, the computer scientist who will not be interested in how we specify **B**, but who will be iteratively computing **Bx** and $\mathbf{B}'\mathbf{y}$ to optimize the model fitting. The lines beginning with `#%` are expanded by Loptran into more verbose and distracting Fortran 90 code. The second line in the module `matmult`, however, is pure Fortran syntax saying that `bb` is a pointer to a real-valued matrix.

To use `matmult`, two calls must be made, the first one

```
call matmult_init( bb)
```

is done by the physical scientist after he or she has prepared the matrix. Most later calls will be done by numerical analysts in solving code like in Chapter 2. These calls look like

```
stat = matmult_lop( adj, add, x, y)
```

where `adj` is the logical variable saying whether we desire the adjoint or the operator itself, and where `add` is a logical variable saying whether we want to accumulate like $\mathbf{y} \leftarrow \mathbf{y} + \mathbf{Bx}$ or whether we want to erase first and thus do $\mathbf{y} \leftarrow \mathbf{Bx}$. The return value `stat` is an integer parameter, mostly useless (unless you want to use it for error

codes).

Operator initialization often allocates memory. To release this memory, you can `call matmult_close()` although in this case nothing really happens.

We split operators into two independent processes, the first is used for geophysical set up while the second is invoked by mathematical library code (introduced in the next chapter) to find the model that best fits the data. Here is why we do so. It is important that the math code contain nothing about the geophysical particulars. This enables us to use the same math code on many different geophysical problems. This concept of "information hiding" arrived late in human understanding of what is desireable in a computer language. This feature alone is valuable enough to warrant upgrading from Fortran 77 to Fortran 90, and likewise from C to C++. Subroutines and functions are the way that new programs use old ones. Object modules are the way that old programs (math solvers) are able to use new ones (geophysical operators).

1.1. FAMILIAR OPERATORS

The simplest and most fundamental linear operators arise when a matrix operator reduces to a simple row or a column.
A **row** is a summation operation.
A **column** is an impulse response.

If the inner loop of a matrix multiply ranges within a
row, the operator is called *sum* or *pull*.
column, the operator is called *spray* or *push*.

A basic aspect of adjointness is that the adjoint of a row matrix operator is a column matrix operator. For example, the row operator $[a,b]$

$$y = [\, a \;\; b \,] \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = a x_1 + b x_2 \tag{1.1}$$

has an adjoint that is two assignments:

$$\begin{bmatrix} \hat{x}_1 \\ \hat{x}_2 \end{bmatrix} = \begin{bmatrix} a \\ b \end{bmatrix} y \tag{1.2}$$

The adjoint of a sum of N terms is a collection of N assignments.

1.1.1. Adjoint derivative

In numerical analysis we represent the derivative of a time function by a finite difference. We do this by subtracting each two neighboring time points and then dividing by the sample interval Δt. This amounts to convolution with the filter $(1,-1)/\Delta t$. Omitting the Δt we express this concept as:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix} = \begin{bmatrix} -1 & 1 & . & . & . & . \\ . & -1 & 1 & . & . & . \\ . & . & -1 & 1 & . & . \\ . & . & . & -1 & 1 & . \\ . & . & . & . & -1 & 1 \\ . & . & . & . & . & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} \tag{1.3}$$

The **filter impulse response** is seen in any column in the middle of the matrix, namely $(1,-1)$. In the transposed matrix, the filter-impulse response is time-reversed to $(-1,1)$. So, mathematically, we can say that the adjoint of the time

derivative operation is the negative time derivative. This corresponds also to the fact that the complex conjugate of $-i\omega$ is $i\omega$. We can also speak of the adjoint of the boundary conditions: we might say that the adjoint of "no boundary condition" is a "specified value" boundary condition. The last row in equation (1.3) is optional. It may seem unnatural to append a null row, but it can be a small convenience (when plotting) to have the input and output be the same size.

Equation (1.3) is implemented by the code in module `igrad1` which does the operator itself (the forward operator) and its adjoint. igrad1
The adjoint code may seem strange. It might seem more natural to code the adjoint to be the negative of the operator itself and then make the special adjustments for the boundaries. The code given, however, is correct and requires no adjustments at the ends. To see why, notice for each value of `i`, the operator itself handles one row of equation (1.3) while for each `i` the adjoint handles one column. That's why coding the adjoint in this way does not require any special work on the ends. The present method of coding reminds us that the adjoint of a sum of N terms is a collection of N assignments.

The Ratfor90 dialect of Fortran allows us to write the inner code of the `igrad1` module more simply and symmetrically using the syntax of C, C++, and Java where

```
module igrad1 {                                         # gradient in one dimension
#%  _lop( xx,  yy)
integer i
do i= 1, size(xx)-1 {
        if( adj) {
                xx(i+1) = xx(i+1) + yy(i)               # resembles equation (1.2)
                xx(i  ) = xx(i  ) - yy(i)
                }
        else
                yy(i) = yy(i) + xx(i+1) - xx(i)         # resembles equation (1.1)
        }
}
```

Back

expressions like `a=a+b` can be written more tersely as `a+=b`. With this, the heart of module `igrad1` becomes

```
if( adj) {   xx(i+1) += yy(i)
             xx(i)   -= yy(i)
           }
else {       yy(i)   += xx(i+1)
             yy(i)   -= xx(i)
           }
```

where we see that each component of the matrix is handled both by the operator and the adjoint. Think about the forward operator "pulling" a sum into `yy(i)`, and think about the adjoint operator "pushing" or "spraying" the impulse `yy(i)` back into `xx()`.

Figure 1.1 illustrates the use of module `igrad1` for each north-south line of a topographic map. We observe that the gradient gives an impression of illumination from a low sun angle. To apply `igrad1` along the 1-axis for each point on the 2-axis of a two-dimensional map, we use the loop

```
do iy=1,ny
      stat = igrad1_lop( adj, add, map(:,iy), ruf(:,iy))
```

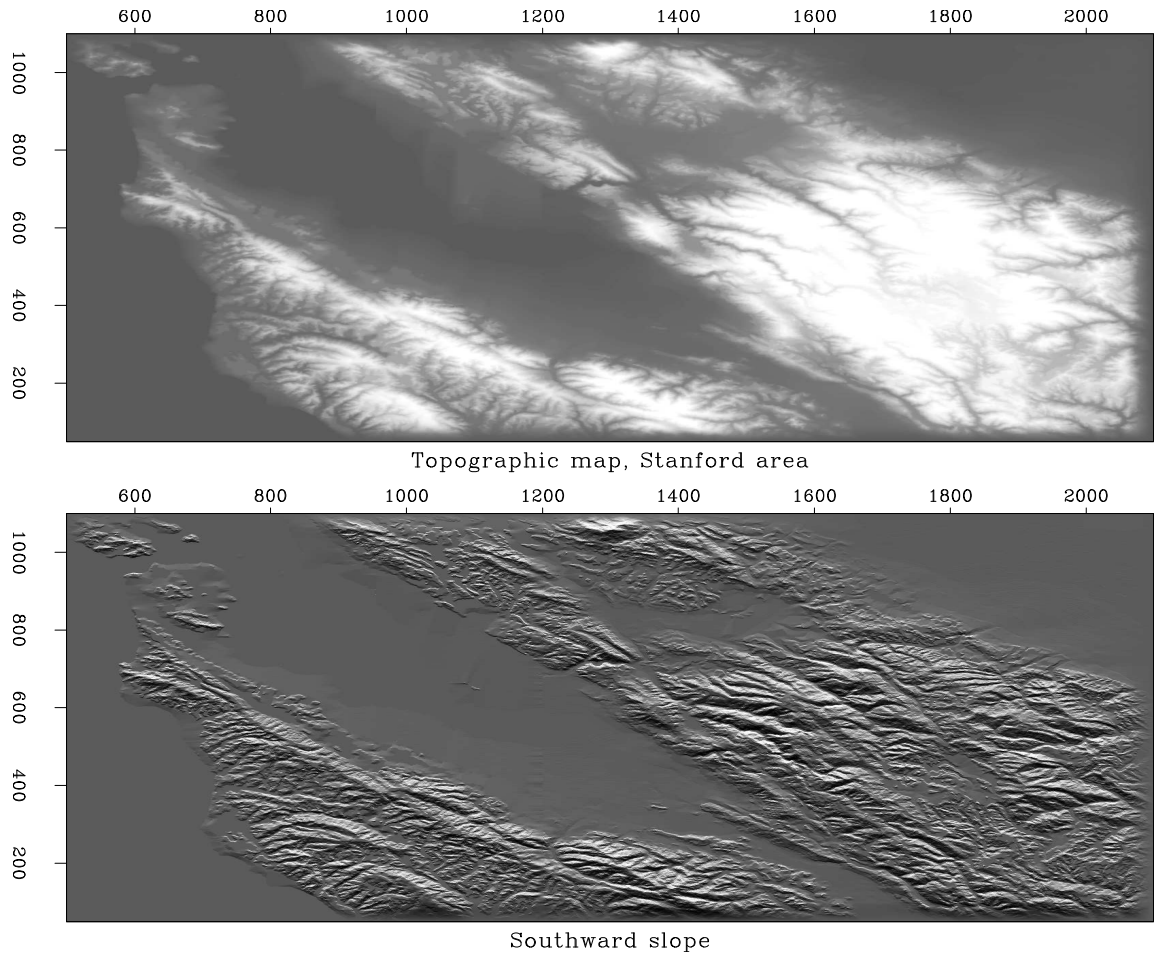


Figure 1.1: Topography near Stanford (top) southward slope (bottom). ajt-stangrad90 [ER,M]

On the other hand, to see the east-west gradient, we use the loop

```
do ix=1,nx
      stat = igrad1_lop( adj, add, map(ix,:), ruf(ix,:))
```

1.1.2. Transient convolution

The next operator we examine is convolution. It arises in many applications; and it could be derived in many ways. A basic derivation is from the multiplication of two polynomials, say $X(Z) = x_1 + x_2 Z + x_3 Z^2 + x_4 Z^3 + x_5 Z^4 + x_6 Z^5$ times $B(Z) = b_1 + b_2 Z + b_3 Z^2 + b_4 Z^3$.[2] Identifying the k-th power of Z in the product

[2] This book is more involved with matrices than with Fourier analysis. If it were more Fourier analysis we would choose notation to begin subscripts from zero like this: $B(Z) = b_0 + b_1 Z + b_2 Z^2 + b_3 Z^3$.

$Y(Z) = B(Z)X(Z)$ gives the k-th row of the convolution transformation (1.4).

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \end{bmatrix} = \begin{bmatrix} b_1 & 0 & 0 & 0 & 0 & 0 \\ b_2 & b_1 & 0 & 0 & 0 & 0 \\ b_3 & b_2 & b_1 & 0 & 0 & 0 \\ 0 & b_3 & b_2 & b_1 & 0 & 0 \\ 0 & 0 & b_3 & b_2 & b_1 & 0 \\ 0 & 0 & 0 & b_3 & b_2 & b_1 \\ 0 & 0 & 0 & 0 & b_3 & b_2 \\ 0 & 0 & 0 & 0 & 0 & b_3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} = \mathbf{Bx} \tag{1.4}$$

Notice that columns of equation (1.4) all contain the same signal, but with different shifts. This signal is called the filter's impulse response.

Equation (1.4) could be rewritten as

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \end{bmatrix} = \begin{bmatrix} x_1 & 0 & 0 \\ x_2 & x_1 & 0 \\ x_3 & x_2 & x_1 \\ x_4 & x_3 & x_2 \\ x_5 & x_4 & x_3 \\ x_6 & x_5 & x_4 \\ 0 & x_6 & x_5 \\ 0 & 0 & x_6 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \mathbf{Xb} \qquad (1.5)$$

In applications we can choose between $\mathbf{y} = \mathbf{Xb}$ and $\mathbf{y} = \mathbf{Bx}$. In one case the output $\mathbf{y}$ is dual to the filter $\mathbf{b}$, and in the other case the output $\mathbf{y}$ is dual to the input $\mathbf{x}$. Sometimes we must solve for $\mathbf{b}$ and sometimes for $\mathbf{x}$; so sometimes we use equation (1.5)

and sometimes (1.4). Such solutions begin from the adjoints. The adjoint of (1.4) is

$$\begin{bmatrix} \hat{x}_1 \\ \hat{x}_2 \\ \hat{x}_3 \\ \hat{x}_4 \\ \hat{x}_5 \\ \hat{x}_6 \end{bmatrix} = \begin{bmatrix} b_1 & b_2 & b_3 & 0 & 0 & 0 & 0 & 0 \\ 0 & b_1 & b_2 & b_3 & 0 & 0 & 0 & 0 \\ 0 & 0 & b_1 & b_2 & b_3 & 0 & 0 & 0 \\ 0 & 0 & 0 & b_1 & b_2 & b_3 & 0 & 0 \\ 0 & 0 & 0 & 0 & b_1 & b_2 & b_3 & 0 \\ 0 & 0 & 0 & 0 & 0 & b_1 & b_2 & b_3 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \end{bmatrix} \tag{1.6}$$

The adjoint ***crosscorrelate**s* with the filter instead of convolving with it (because the filter is backwards). Notice that each row in equation (1.6) contains all the filter coefficients and there are no rows where the filter somehow uses zero values off the ends of the data as we saw earlier. In some applications it is important not to assume zero values beyond the interval where inputs are given.

The adjoint of (1.5) crosscorrelates a fixed portion of filter input across a vari-

able portion of filter output.

$$\begin{bmatrix} \hat{b}_1 \\ \hat{b}_2 \\ \hat{b}_3 \end{bmatrix} = \begin{bmatrix} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & 0 & 0 \\ 0 & x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & 0 \\ 0 & 0 & x_1 & x_2 & x_3 & x_4 & x_5 & x_6 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \end{bmatrix} \tag{1.7}$$

Module `tcai1` is used for $\mathbf{y} = \mathbf{Bx}$ and module `tcaf1` is used for $\mathbf{y} = \mathbf{Xb}$. tcai1 tcaf1

The polynomials $X(Z)$, $B(Z)$, and $Y(Z)$ are called Z transforms. An important fact in real life (but not important here) is that the Z transforms are Fourier transforms in disguise. Each polynomial is a sum of terms and the sum amounts to a Fourier sum when we take $Z = e^{i\omega\Delta t}$. The very expression $Y(Z) = B(Z)X(Z)$ says that a product in the frequency domain (Z has a numerical value) is a convolution in the time domain (that's how we multipy polynomials, convolve their coefficients).

```
module tcai1 {            # Transient Convolution Adjoint Input 1-D. yy(m1+n1)
real, dimension (:), pointer :: bb
#%  _init( bb)
#%  _lop ( xx, yy)
integer b, x, y
if( size(yy) < size (xx) + size(bb) - 1 ) call erexit('tcai')
do b= 1, size(bb) {
do x= 1, size(xx) {                  y = x + b - 1
        if( adj)        xx(x) +=  yy(y) * bb(b)
        else            yy(y) +=  xx(x) * bb(b)
        }}
}
```

Back

```
module tcaf1 {              # Transient Convolution, Adjoint is the Filter, 1-D
real, dimension (:), pointer :: xx
#%  _init( xx)
#%  _lop ( bb,  yy)
integer         x,    b,    y
if( size(yy) < size(xx) + size(bb) - 1 )   call erexit('tcaf')
do b= 1, size(bb) {
do x= 1, size(xx) {                  y = x + b - 1
        if( adj)        bb(b) +=  yy(y) * xx(x)
        else            yy(y) +=  bb(b) * xx(x)
        } }
}
```

Back

1.1.3. Internal convolution

%sxconvolution ! internal Convolution is the computational equivalent of ordinary linear differential operators (with constant coefficients). Applications are vast, and end effects are important. Another choice of data handling at ends is that zero data not be assumed beyond the interval where the data is given. This is important in data where the crosscorrelation changes with time. Then it is sometimes handled as constant in short time windows. Care must be taken that zero signal values not be presumed off the ends of those short time windows; otherwise, the many ends of the many short segments can overwhelm the results.

In the sets (1.4) and (1.5), the top two equations explicitly assume that the input data vanishes before the interval on which it is given, and likewise at the bottom. Abandoning the top two and bottom two equations in (1.5) we get:

$$\begin{bmatrix} y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix} = \begin{bmatrix} x_3 & x_2 & x_1 \\ x_4 & x_3 & x_2 \\ x_5 & x_4 & x_3 \\ x_6 & x_5 & x_4 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{1.8}$$

The adjoint is

$$\begin{bmatrix} \hat{b}_1 \\ \hat{b}_2 \\ \hat{b}_3 \end{bmatrix} = \begin{bmatrix} x_3 & x_4 & x_5 & x_6 \\ x_2 & x_3 & x_4 & x_5 \\ x_1 & x_2 & x_3 & x_4 \end{bmatrix} \begin{bmatrix} y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix} \tag{1.9}$$

The difference between (1.9) and (1.7) is that here the adjoint crosscorrelates a fixed portion of *output* across a variable portion of *input*, whereas with (1.7) the adjoint crosscorrelates a fixed portion of *input* across a variable portion of *output*.

In practice we typically allocate equal space for input and output. Because the output is shorter than the input, it could slide around in its allocated space, so its location is specified by an additional parameter called its `lag`. icaf1 The value of `lag` always used in this book is `lag=1`. For `lag=1` the module `icaf1` implements not

```
module icaf1 {                  # Internal Convolution, Adjoint is Filter. 1-D
integer :: lag
real, dimension (:), pointer :: xx
#%  _init ( xx, lag)
#%  _lop  ( bb,  yy)
integer   x,  b,   y
do b= 1, size(bb) {
        do y= 1+size(bb)-lag, size(yy)-lag+1 {    x= y - b + lag
                if( adj)        bb(b) +=  yy(y) * xx(x)
                else            yy(y) +=  bb(b) * xx(x)
                }
        }
}
```

Back

equation (1.8) but (1.10):

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ x_3 & x_2 & x_1 \\ x_4 & x_3 & x_2 \\ x_5 & x_4 & x_3 \\ x_6 & x_5 & x_4 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{1.10}$$

It may seem a little odd to put the required zeros at the beginning of the output, but filters are generally designed so that their strongest coefficient is the first, namely `bb(1)` so the alignment of input and output in equation (1.10) is the most common one.

The **end effect**s of the convolution modules are summarized in Figure 1.2.

1.1.4. Zero padding is the transpose of truncation

Surrounding a dataset by zeros (**zero pad**ding) is adjoint to throwing away the extended data (**truncation**). Let us see why this is so. Set a signal in a vector **x**, and

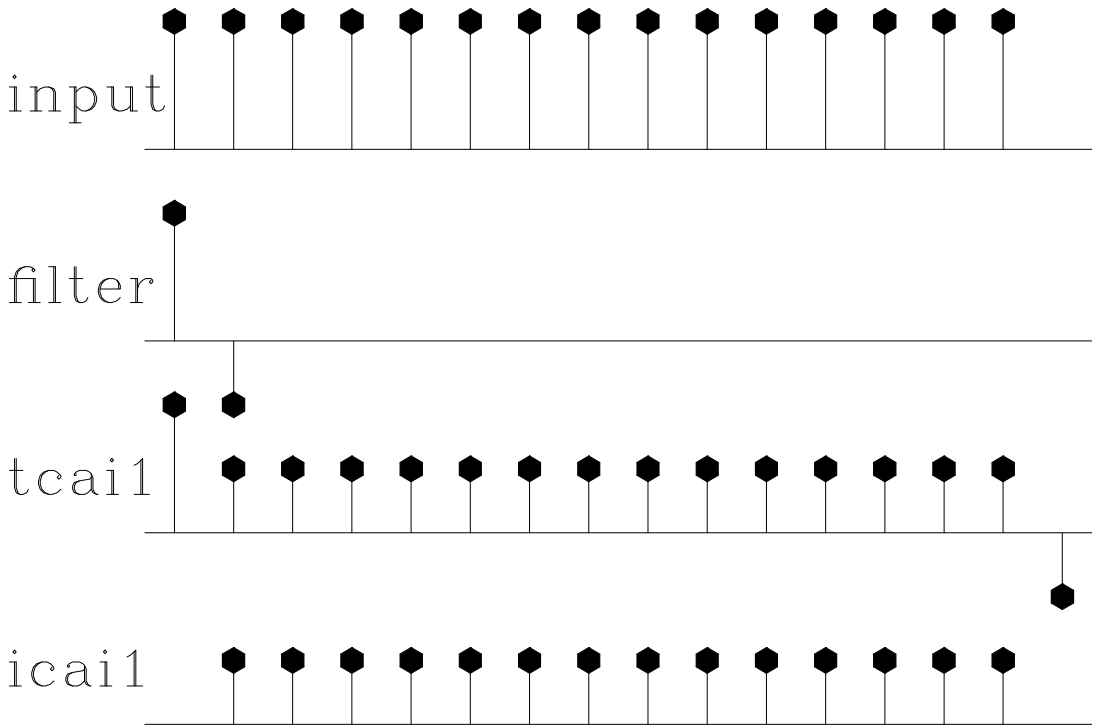


Figure 1.2: Example of convolution end-effects. From top to bottom: input; filter; output of `tcai1()`; output of `icaf1()` also with (`lag=1`). ajt-conv90 [ER]

then to make a longer vector **y**, add some zeros at the end of **x**. This zero padding can be regarded as the matrix multiplication

$$\mathbf{y} \quad = \quad \begin{bmatrix} \mathbf{I} \\ \mathbf{0} \end{bmatrix} \mathbf{x} \tag{1.11}$$

The matrix is simply an identity matrix **I** above a zero matrix **0**. To find the transpose to zero-padding, we now transpose the matrix and do another matrix multiply:

$$\tilde{\mathbf{x}} \quad = \quad \begin{bmatrix} \mathbf{I} & \mathbf{0} \end{bmatrix} \mathbf{y} \tag{1.12}$$

So the transpose operation to zero padding data is simply *truncating* the data back to its original length. Module `zpad1` below pads zeros on both ends of its input. Modules for two- and three-dimensional padding are in the library named `zpad2()` and `zpad3()`. zpad1

```
module zpad1 {                          # Zero pad.  Surround data by zeros. 1-D
#%  _lop( data,  padd)
integer                                 p,  d
do d= 1, size(data) {                   p = d + (size(padd)-size(data))/2
        if( adj)
                        data(d) = data(d) + padd(p)
        else
                        padd(p) = padd(p) + data(d)
        }
}
```

Back

1.1.5. Adjoints of products are reverse-ordered products of adjoints

Here we examine an example of the general idea that adjoints of products are reverse-ordered products of adjoints. For this example we use the Fourier transformation. No details of **Fourier transformation** are given here and we merely use it as an example of a square matrix $\mathbf{F}$. We denote the complex-conjugate transpose (or **adjoint**) matrix with a prime, i.e., $\mathbf{F}'$. The adjoint arises naturally whenever we consider energy. The statement that Fourier transforms conserve energy is $\mathbf{y}'\mathbf{y} = \mathbf{x}'\mathbf{x}$ where $\mathbf{y} = \mathbf{F}\mathbf{x}$. Substituting gives $\mathbf{F}'\mathbf{F} = \mathbf{I}$, which shows that the inverse matrix to Fourier transform happens to be the complex conjugate of the transpose of $\mathbf{F}$.

With Fourier transforms, **zero pad**ding and **truncation** are especially prevalent. Most modules transform a dataset of length of 2^n, whereas dataset lengths are often of length $m \times 100$. The practical approach is therefore to pad given data with zeros. Padding followed by Fourier transformation $\mathbf{F}$ can be expressed in matrix algebra as

$$\text{Program} \quad = \quad \mathbf{F} \begin{bmatrix} \mathbf{I} \\ \mathbf{0} \end{bmatrix} \tag{1.13}$$

According to matrix algebra, the transpose of a product, say $\mathbf{AB}=\mathbf{C}$, is the product $\mathbf{C}'=\mathbf{B}'\mathbf{A}'$ in reverse order. So the adjoint routine is given by

$$\text{Program}' \quad = \quad \left[\begin{array}{cc} \mathbf{I} & \mathbf{0} \end{array} \right] \mathbf{F}' \tag{1.14}$$

Thus the adjoint routine *truncates* the data *after* the inverse Fourier transform. This concrete example illustrates that common sense often represents the mathematical abstraction that adjoints of products are reverse-ordered products of adjoints. It is also nice to see a formal mathematical notation for a practical necessity. Making an approximation need not lead to collapse of all precise analysis.

1.1.6. Nearest-neighbor coordinates

In describing physical processes, we often either specify models as values given on a uniform mesh or we record data on a uniform mesh. Typically we have a function f of time t or depth z and we represent it by `f(iz)` corresponding to $f(z_i)$ for $i=1,2,3,\ldots,n_z$ where $z_i=z_0+(i-1)\Delta z$. We sometimes need to handle depth as an integer counting variable i and we sometimes need to handle it as a floating-point variable z. Conversion from the counting variable to the floating-point variable is exact and is often seen in a computer idiom such as either of

```
do iz= 1, nz {   z = z0 + (iz-1) * dz
do i3= 1, n3 {  x3 = o3 + (i3-1) * d3
```

The reverse conversion from the floating-point variable to the counting variable is inexact. The easiest thing is to place it at the nearest neighbor. This is done by solving for `iz`, then adding one half, and then rounding down to the nearest integer. The familiar computer idioms are:

```
iz =  .5 + 1 + ( z - z0) / dz
iz = 1.5 +     ( z - z0) / dz
i3 = 1.5 +     (x3 - o3) / d3
```

A small warning is in order: People generally use positive counting variables. If you also include negative ones, then to get the nearest integer, you should do your rounding with the Fortran function `NINT()`.

1.1.7. Data-push binning

Binning is putting data values in bins. Nearest-neighbor binning is an operator. There is both a forward operator and its adjoint. Normally the model consists of values given on a uniform mesh, and the data consists of pairs of numbers (ordinates

at coordinates) sprinkled around in the continuum (although sometimes the data is uniformly spaced and the model is not).

In both the forward and the adjoint operation, each data coordinate is examined and the nearest mesh point (the bin) is found. For the forward operator, the value of the bin is added to that of the data. The adjoint is the reverse: we add the value of the data to that of the bin. Both are shown in two dimensions in module `bin2`. bin2 The most typical application requires an additional step, inversion. In the inversion applications each bin contains a different number of data values. After the adjoint operation is performed, the inverse operator divides the bin value by the number of points in the bin. It is this inversion operator that is generally called binning. To find the number of data points in a bin, we can simply apply the adjoint of `bin2` to pseudo data of all ones. To capture this idea in an equation, let $\mathbf{B}$ denote the linear operator in which the bin value is sprayed to the data values. The inverse operation, in which the data values in the bin are summed and divided by the number in the bin, is represented by

$$\mathbf{m} \quad = \quad \mathbf{diag}(\mathbf{B}'\mathbf{1})^{-1}\mathbf{B}'\mathbf{d} \tag{1.15}$$

Empty bins, of course, leave us a problem. That we'll address in chapter 3. In

```
module bin2 {
# Data-push binning in 2-D.
integer :: m1, m2
real    ::  o1,d1,o2,d2
real, dimension (:,:), pointer :: xy
#% _init(     m1,m2, o1,d1,o2,d2,xy)
#% _lop ( mm (m1,m2),  dd (:))
integer    i1,i2, id
do id=1,size(dd) {
        i1 = 1.5 + (xy(id,1)-o1)/d1
        i2 = 1.5 + (xy(id,2)-o2)/d2
        if( 1<=i1 && i1<=m1 &&
            1<=i2 && i2<=m2   )
                if( adj)
                        mm(i1,i2) = mm(i1,i2) +  dd( id)
                else
                        dd( id)   = dd( id)   +  mm(i1,i2)
        }
}
```

Back

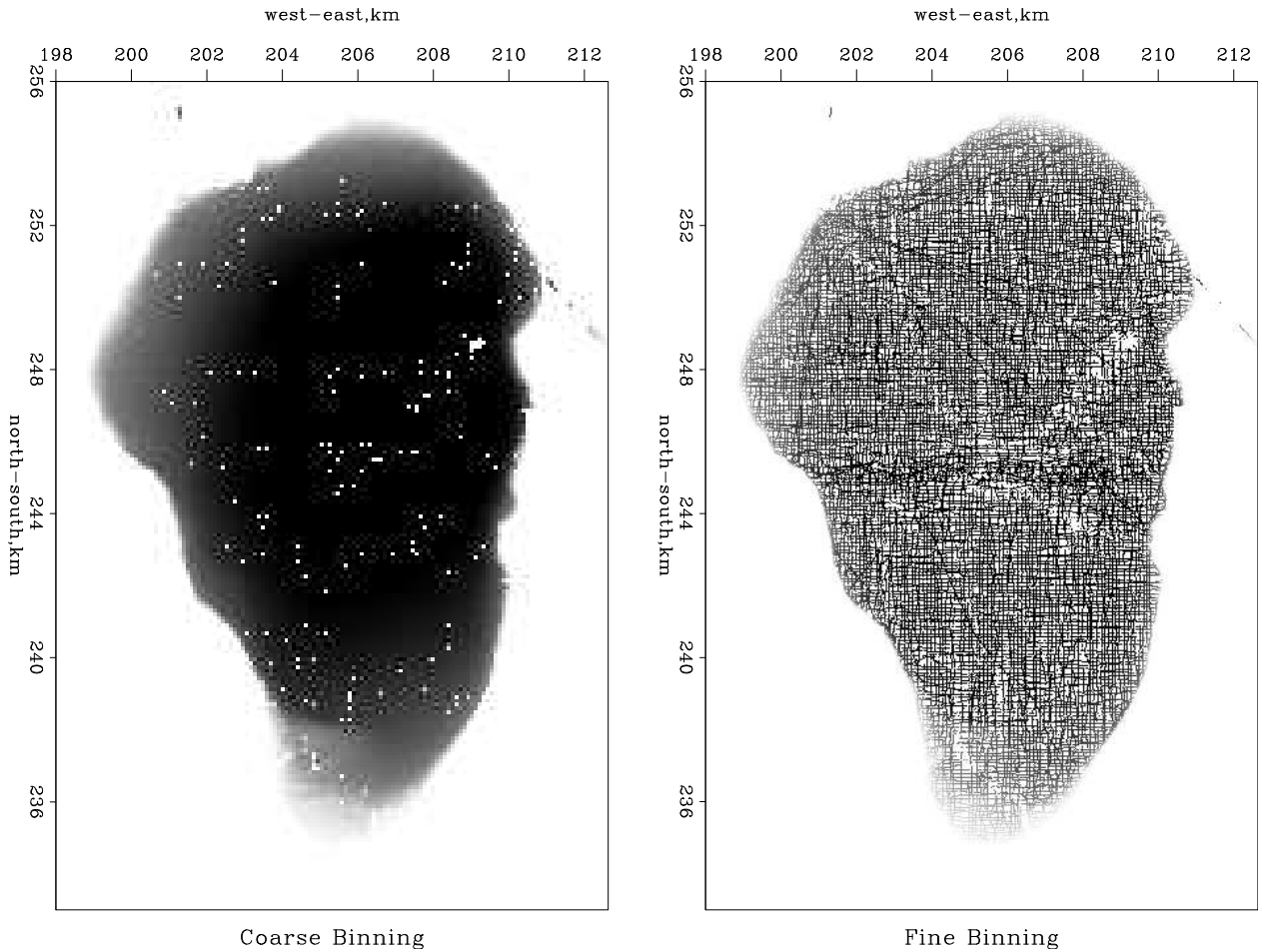


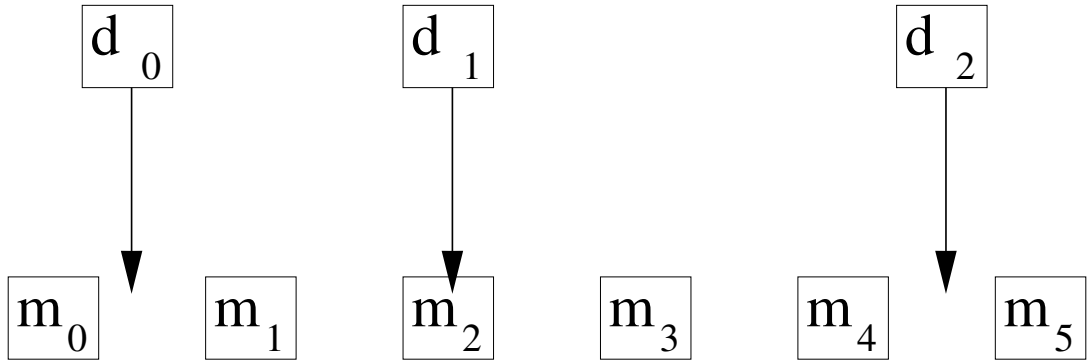
Figure 1.3: Binned depths of the Sea of Galilee. ajt-galbin90 [ER]

Figure 1.3, the empty bins contain zero values.

1.1.8. Linear interpolation

The **linear interpolation** operator is much like the binning operator but a little fancier. When we perform the forward operation, we take each data coordinate and see which two model bin centers bracket it. Then we pick up the two bracketing model values and weight each of them in proportion to their nearness to the data coordinate, and add them to get the data value (ordinate). The adjoint operation is adding a data value back into the model vector; using the same two weights, the adjoint distributes the data ordinate value between the two nearest bins in the model vector. For example, suppose we have a data point near each end of the model and a third data point exactly in the middle. Then for a model space 6 points long, as shown in Figure 1.4, we have the operator in (1.16).

Figure 1.4: Uniformly sampled model space and irregularly sampled data space corresponding to (1.16). ajt-helgerud [NR]



$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix} \approx \begin{bmatrix} .7 & .3 & . & . & . & . \\ . & . & 1 & . & . & . \\ . & . & . & . & .5 & .5 \end{bmatrix} \begin{bmatrix} m_0 \\ m_1 \\ m_2 \\ m_3 \\ m_4 \\ m_5 \end{bmatrix} \tag{1.16}$$

The two weights in each row sum to unity. If a binning operator were used for the same data and model, the binning operator would contain a "1." in each row. In one dimension (as here), data coordinates are often sorted into sequence, so that the matrix is crudely a diagonal matrix like equation (1.16). If the data coordinates covered the model space uniformly, the adjoint would roughly be the inverse. Oth-

erwise, when data values pile up in some places and gaps remain elsewhere, the adjoint would be far from the inverse.

Module `lint1` does linear interpolation and its adjoint. In chapters 3 and 6 we build inverse operators. lint1

1.1.9. Spray and sum : scatter and gather

Perhaps the most common operation is the summing of many values to get one value. Its adjoint operation takes a single input value and throws it out to a space of many values. The **summation operator** is a row vector of ones. Its adjoint is a column vector of ones. In one dimension this operator is almost too easy for us to bother showing a routine. But it is more interesting in three dimensions, where we could be summing or spraying on any of three subscripts, or even summing on some and spraying on others. In module `spraysum`, both input and output are taken to be three-dimensional arrays. Externally, however, either could be a scalar, vector, plane, or cube. For example, the internal array `xx(n1,1,n3)` could be externally the matrix `map(n1,n3)`. When module `spraysum` is given the input dimensions and output dimensions stated below, the operations stated alongside are implied.

```
# Nearest-neighbor interpolation would do this:  data = model( 1.5 + (t-t0)/dt)
#                               This is likewise but with _linear_ interpolation.
module lint1 {
real :: o1,d1
real, dimension (:), pointer :: coordinate
#%  _init ( o1,d1, coordinate)
#%  _lop  ( mm, dd)
integer i, im,  id
real    f, fx,gx
do id= 1, size(dd) {
        f = (coordinate(id)-o1)/d1;     i=f  ;   im= 1+i
        if( 1<=im && im< size(mm)) {   fx=f-i;   gx= 1.-fx
                if( adj) {
                        mm(im  ) +=  gx * dd(id)
                        mm(im+1) +=  fx * dd(id)
                        }
                else
                        dd(id)   +=  gx * mm(im)  +  fx * mm(im+1)
                }
        }
}
```

Back

`(n1,n2,n3)`	`(1,1,1)`	Sum a cube into a value.
`(1,1,1)`	`(n1,n2,n3)`	Spray a value into a cube.
`(n1,1,1)`	`(n1,n2,1)`	Spray a column into a matrix.
`(1,n2,1)`	`(n1,n2,1)`	Spray a row into a matrix.
`(n1,n2,1)`	`(n1,n2,n3)`	Spray a plane into a cube.
`(n1,n2,1)`	`(n1,1,1)`	Sum rows of a matrix into a column.
`(n1,n2,1)`	`(1,n2,1)`	Sum columns of a matrix into a row.
`(n1,n2,n3)`	`(n1,n2,n3)`	Copy and add the whole cube.

If an axis is not of unit length on either input or output, then both lengths must be the same; otherwise, there is an error. Normally, after (possibly) erasing the output, we simply loop over all points on each axis, adding the input to the output. This implements either a copy or an add, depending on the `add` parameter. It is either a spray, a sum, or a copy, according to the specified axis lengths. spraysum

```
module spraysum {                       # Spray or sum over 1, 2, and/or 3-axis.
integer ::   n1,n2,n3,     m1,m2,m3
#%  _init(   n1,n2,n3,     m1,m2,m3)
#%  _lop( xx(n1,n2,n3), yy(m1,m2,m3))
integer i1,i2,i3,  x1,x2,x3, y1,y2,y3
        if( n1 != 1  &&  m1 != 1  &&  n1 != m1)   call erexit('spraysum: n1,m1')
        if( n2 != 1  &&  m2 != 1  &&  n2 != m2)   call erexit('spraysum: n2,m2')
        if( n3 != 1  &&  m3 != 1  &&  n3 != m3)   call erexit('spraysum: n3,m3')
do i3= 1, max0(n3,m3) {   x3= min0(i3,n3);   y3= min0(i3,m3)
do i2= 1, max0(n2,m2) {   x2= min0(i2,n2);   y2= min0(i2,m2)
do i1= 1, max0(n1,m1) {   x1= min0(i1,n1);   y1= min0(i1,m1)
        if( adj)  xx(x1,x2,x3)  +=  yy(y1,y2,y3)
        else      yy(y1,y2,y3)  +=  xx(x1,x2,x3)
        }}}
}
```

Back

1.1.10. Causal and leaky integration

Causal integration is defined as

$$y(t) \quad = \quad \int_{-\infty}^{t} x(\tau)\, d\tau \tag{1.17}$$

Leaky integration is defined as

$$y(t) \quad = \quad \int_{0}^{\infty} x(t-\tau)\, e^{-\alpha\tau}\, d\tau \tag{1.18}$$

As $\alpha \to 0$, leaky integration becomes causal integration. The word "leaky" comes from electrical circuit theory where the voltage on a capacitor would be the integral of the current if the capacitor did not leak electrons.

Sampling the time axis gives a matrix equation that we should call causal summation, but we often call it causal integration. Equation (1.19) represents causal

integration for $\rho = 1$ and leaky integration for $0 < \rho < 1$.

$$\mathbf{y} \quad = \quad \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix} \quad = \quad \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ \rho & 1 & 0 & 0 & 0 & 0 & 0 \\ \rho^2 & \rho & 1 & 0 & 0 & 0 & 0 \\ \rho^3 & \rho^2 & \rho & 1 & 0 & 0 & 0 \\ \rho^4 & \rho^3 & \rho^2 & \rho & 1 & 0 & 0 \\ \rho^5 & \rho^4 & \rho^3 & \rho^2 & \rho & 1 & 0 \\ \rho^6 & \rho^5 & \rho^4 & \rho^3 & \rho^2 & \rho & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} \quad = \quad \mathbf{Cx} \tag{1.19}$$

(The discrete world is related to the continuous by $\rho = e^{-\alpha \Delta \tau}$ and in some applications, the diagonal is 1/2 instead of 1.) Causal integration is the simplest prototype of a recursive operator. The coding is trickier than that for the operators we considered earlier. Notice when you compute y_5 that it is the sum of 6 terms, but that this sum is more quickly computed as $y_5 = \rho y_4 + x_5$. Thus equation (1.19) is more efficiently thought of as the recursion

$$y_t \quad = \quad \rho \; y_{t-1} + x_t \qquad\qquad t \text{ increasing} \tag{1.20}$$

(which may also be regarded as a numerical representation of the **differential equation** $dy/dt + y(1-\rho)/\Delta t = x(t)$.)

When it comes time to think about the adjoint, however, it is easier to think of equation (1.19) than of (1.20). Let the matrix of equation (1.19) be called $\mathbf{C}$. Transposing to get $\mathbf{C}'$ and applying it to $\mathbf{y}$ gives us something back in the space of $\mathbf{x}$, namely $\tilde{\mathbf{x}} = \mathbf{C}'\mathbf{y}$. From it we see that the adjoint calculation, if done recursively, needs to be done backwards, as in

$$\tilde{x}_{t-1} \quad = \quad \rho \tilde{x}_t + y_{t-1} \qquad\qquad t \text{ decreasing} \tag{1.21}$$

Thus the adjoint of causal integration is **anticausal integration**.

A module to do these jobs is `leakint`. The code for anticausal integration is not obvious from the code for integration and the adjoint coding tricks we learned earlier. To understand the adjoint, you need to inspect the detailed form of the expression $\tilde{\mathbf{x}} = \mathbf{C}'\mathbf{y}$ and take care to get the ends correct. Figure 1.5 illustrates the program for $\rho = 1$. leakint

Later we will consider equations to march wavefields up towards the earth's surface, a layer at a time, an operator for each layer. Then the adjoint will start from the earth's surface and march down, a layer at a time, into the earth.

```
module leakint {                                                # leaky integration
real ::    rho
#%  _init( rho)
#%  _lop ( xx,  yy)
integer i, n
real tt
n = size (xx); tt = 0.
if( adj)
        do i= n, 1, -1 {   tt = rho*tt + yy(i)
                xx(i)  +=  tt
                }
else
        do i= 1, n {       tt  = rho*tt + xx(i)
                yy(i)  +=  tt
                }
}
```

Back

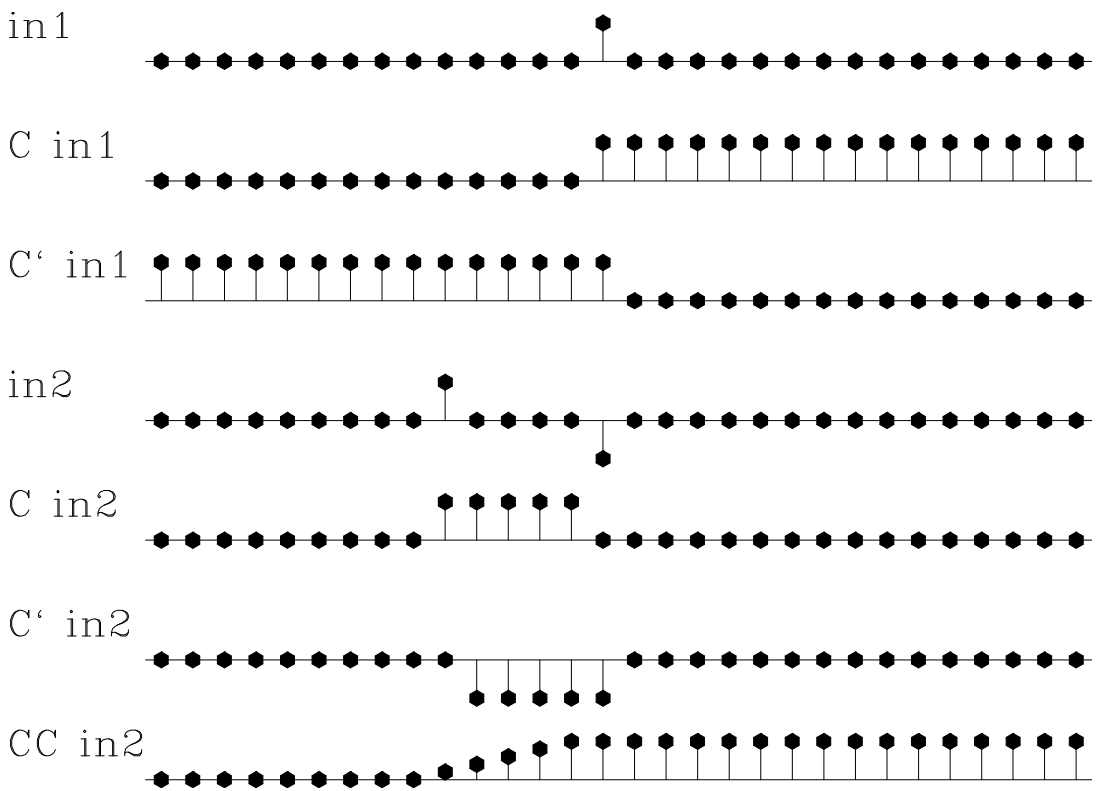


Figure 1.5: `in1` is an input pulse. `C in1` is its causal integral. `C' in1` is the anticausal integral of the pulse. `in2` is a separated doublet. Its causal integration is a box and its anticausal integration is a negative box. `CC in2` is the double causal integral of `in2`. How can an equilateral triangle be built? ajt-causint90 [ER]

EXERCISES:

1 Consider the matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & \rho & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.22}$$

and others like it with ρ in other locations. Show what combination of these matrices will represent the leaky integration matrix in equation (1.19). What is the adjoint?

2 Modify the calculation in Figure 1.5 so that there is a triangle waveform on the bottom row.

3 Notice that the triangle waveform is not time aligned with the input `in2`. Force time alignment with the operator $\mathbf{C}'\mathbf{C}$ or $\mathbf{C}\mathbf{C}'$.

4 Modify `leakint` /prog:leakint by changing the diagonal to contain 1/2 instead of 1. Notice how time alignment changes in Figure 1.5.

1.1.11. Backsolving, polynomial division and deconvolution

Ordinary differential equations often lead us to the backsolving operator. For example, the damped harmonic oscillator leads to a special case of equation (1.23) where $(a_3, a_4, \cdots) = 0$. There is a huge literature on finite-difference solutions of ordinary differential equations that lead to equations of this type. Rather than derive such an equation on the basis of many possible physical arrangements, we can begin from the filter transformation $\mathbf{B}$ in (1.4) but put the matrix on the other side of the equation so our transformation can be called one of inversion or backsubstitution. Let us also force the matrix $\mathbf{B}$ to be a square matrix by truncating it with $\mathbf{T} = [\mathbf{I} \quad \mathbf{0}]$, say $\mathbf{A} = [\mathbf{I} \quad \mathbf{0}]\mathbf{B} = \mathbf{TB}$. To link up with applications in later chapters, I specialize to

1's on the main diagonal and insert some bands of zeros.

$$\mathbf{Ay} \quad = \quad \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ a_1 & 1 & 0 & 0 & 0 & 0 & 0 \\ a_2 & a_1 & 1 & 0 & 0 & 0 & 0 \\ 0 & a_2 & a_1 & 1 & 0 & 0 & 0 \\ 0 & 0 & a_2 & a_1 & 1 & 0 & 0 \\ a_5 & 0 & 0 & a_2 & a_1 & 1 & 0 \\ 0 & a_5 & 0 & 0 & a_2 & a_1 & 1 \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix} \quad = \quad \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} \quad = \quad \mathbf{x} \tag{1.23}$$

Algebraically, this operator goes under the various names, "backsolving", "polynomial division", and "deconvolution". The leaky integration transformation (1.19) is a simple example of backsolving when $a_1 = -\rho$ and $a_2 = a_5 = 0$. To confirm this, you need to verify that the matrices in (1.23) and (1.19) are mutually inverse.

A typical row in equation (1.23) says

$$x_t \quad = \quad y_t + \sum_{\tau > 0} a_\tau \; y_{t-\tau} \tag{1.24}$$

Change the signs of all terms in equation (1.24) and move some terms to the opposite side

$$y_t \quad = \quad x_t \; - \sum_{\tau>0} a_\tau \; y_{t-\tau} \tag{1.25}$$

Equation (1.25) is a recursion to find y_t from the values of y at earlier times.

In the same way that equation (1.4) can be interpreted as $Y(Z) = B(Z)X(Z)$, equation (1.23) can be interpreted as $A(Z)Y(Z) = X(Z)$ which amounts to $Y(Z) = X(Z)/A(Z)$. Thus, convolution is amounts to polynomial multiplication while the backsubstitution we are doing here is called deconvolution, and it amounts to polynomial division.

A causal operator is one that uses its present and past inputs to make its current output. Anticausal operators use the future but not the past. Causal operators are generally associated with lower triangular matrices and positive powers of Z, whereas anticausal operators are associated with upper triangular matrices and negative powers of Z. A transformation like equation (1.23) but with the transposed matrix would require us to run the recursive solution the opposite direction in time, as we did with leaky integration.

A module to backsolve (1.23) is `polydiv1`. polydiv1

```
module polydiv1 {                    # Polynomial division (recursive filtering)
real, dimension (:), pointer :: aa
#%  _init ( aa)
#%  _lop  ( xx, yy)
integer  ia, ix, iy
real     tt
if( adj)
        do ix= size(xx), 1, -1 {
                tt = yy( ix)
                do ia = 1, min( size(aa), size (xx) - ix) {
                        iy = ix + ia
                        tt -=  aa( ia) * xx( iy)
                        }
                xx( ix) = xx( ix) + tt
                }
else
        do iy= 1, size(xx) {
                tt = xx( iy)
                do ia = 1, min( size(aa), iy-1) {
                        ix = iy - ia
                        tt -=  aa( ia) * yy( ix)
                        }
                yy( iy) =  yy( iy) + tt
                }
}
```

Back

The more complicated an operator, the more complicated is its adjoint. Given a transformation from $\mathbf{x}$ to $\mathbf{y}$ that is $\mathbf{TBy} = \mathbf{x}$, we may wonder if the adjoint transform really is $(\mathbf{TB})'\hat{\mathbf{x}} = \mathbf{y}$. It amounts to asking if the adjoint of $\mathbf{y} = (\mathbf{TB})^{-1}\mathbf{x}$ is $\hat{\mathbf{x}} = ((\mathbf{TB})')^{-1}\mathbf{y}$. Mathematically we are asking if the inverse of a transpose is the transpose of the inverse. This is so because in $\mathbf{AA}^{-1} = \mathbf{I} = \mathbf{I}' = (\mathbf{A}^{-1})'\mathbf{A}'$ the parenthesized object must be the inverse of its neighbor $\mathbf{A}'$.

The adjoint has a meaning which is nonphysical. It is like the forward operator except that we must begin at the final time and revert towards the first. The adjoint pendulum damps as we compute it backward in time, but that, of course, means that the adjoint pendulum diverges as it is viewed moving forward in time.

1.1.12. The basic low-cut filter

Many geophysical measurements contain very low-frequency noise called "drift." For example, it might take some months to survey the depth of a lake. Meanwhile, rainfall or evaporation could change the lake level so that new survey lines become inconsistent with old ones. Likewise, gravimeters are sensitive to atmospheric pressure, which changes with the weather. A magnetic survey of an archeological site

would need to contend with the fact that the earth's main magnetic field is changing randomly through time while the survey is being done. Such noises are sometimes called "secular noise."

The simplest way to eliminate low frequency noise is to take a time derivative. A disadvantage is that the derivative changes the waveform from a pulse to a doublet (finite difference). Here we examine the most basic low-cut filter. It preserves the waveform at high frequencies; it has an adjustable parameter for choosing the bandwidth of the low cut; and it is causal (uses the past but not the future).

We make our causal lowcut filter (highpass filter) by two stages which can be done in either order.

1. Apply a time derivative, actually a finite difference, convolving the data with $(1, -1)$.

2. Integrate, actually to do a leaky integration, to deconvolve with $(1, -\rho)$ where numerically, ρ is slightly less than unity.

The convolution ensures that the zero frequency is removed. The leaky integration almost undoes the differentiation (but does not restore the zero frequency). Adjusting the numerical value of ρ adjusts the cutoff frequency of the filter. To

learn the impulse response of the combined processes, we need to convolve the finite difference $(1,-1)$ with the leaky integration $(1,\rho,\rho^2,\rho^3,\rho^4,\cdots)$. The result is $(1,\rho,\rho^2,\rho^3,\rho^4,\cdots)$ minus $(0,1,\rho,\rho^2,\rho^3,\cdots)$. We can think of this as $(1,0,0,0,0,\cdots)$ minus $(1-\rho)(1,\rho,\rho^2,\rho^3,\cdots)$. In other words the impulse response is an impulse followed by the negative of a weak $(1-\rho)$ decaying exponential ρ^t. Roughly speaking, the cutoff frequency of the filter corresponds to matching one wavelength to the exponential decay time.

Some exercise with Fourier transforms or Z-transforms[3], shows the Fourier transform of this highpass filter to be

$$H(Z) \quad = \quad \frac{1-Z}{1-\rho Z} \quad = \quad 1-(1-\rho)[Z^1+\rho Z^2+\rho^2 Z^3+\rho^3 Z^4\cdots] \quad (1.26)$$

where the unit-delay operator is $Z=e^{i\omega\Delta t}$ and where ω is the frequency. A symmetrical (noncausal) lowcut filter would filter once forward with $H(Z)$ and once backwards (adjoint) with $H(1/Z)$. This is not the place for a detailed Fourier analysis of this filter but it is the place to mention that a cutoff filter is typically specified

[3] An introduction to Z-transforms is found in my earlier books, FGDP and ESA-PVI.

by its cutoff frequency, a frequency that separates the pass and reject region. For this filter, the cutoff frequency ω_0 would correspond to matching a quarter wavelength of a sinusoid to the exponential decay length of ρ^k, namely, say the value of k for which $\rho^k \approx 1/2$

Seismological data is more complex. A single "measurement" consists of an explosion and echo signals recorded at many locations. As before, a complete survey is a track (or tracks) of explosion locations. Thus, in seismology, data space is higher dimensional. Its most troublesome noise is not simply low frequency; it is low velocity. We will do more with multidimensional data in later chapters.

EXERCISES:

1 Give an analytic expression for the waveform of equation (1.26).

2 Define a low-pass filter as $1 - H(Z)$. What is the low-pass impulse response?

3 Put Galilee data on a coarse mesh. Consider north-south lines as one-dimensional signals. Find the value of ρ for which H is the most pleasing filter.

4 Find the value of ρ for which $\bar{H}H$ is the most pleasing filter.

5 Find the value of ρ for which H applied to Galilee has minimum energy. (Experiment with a range of about ten values around your favorite value.)

6 Find the value of ρ for which $\bar{H}H$ applied to Galilee has minimum energy.

7 Repeat above for east-west lines.

1.1.13. SMOOTHING WITH BOX AND TRIANGLE

Simple "**smoothing**" is a common application of filtering. A smoothing filter is one with all positive coefficients. On the time axis, smoothing is often done with a single-pole damped exponential function. On space axes, however, people generally prefer a symmetrical function. We will begin with rectangle and triangle functions. When the function width is chosen to be long, then the computation time can be large, but recursion can shorten it immensely.

The inverse of any polynomial reverberates forever, although it might drop off fast enough for any practical need. On the other hand, a rational filter can suddenly drop to zero and stay there. Let us look at a popular rational filter, the rectangle or

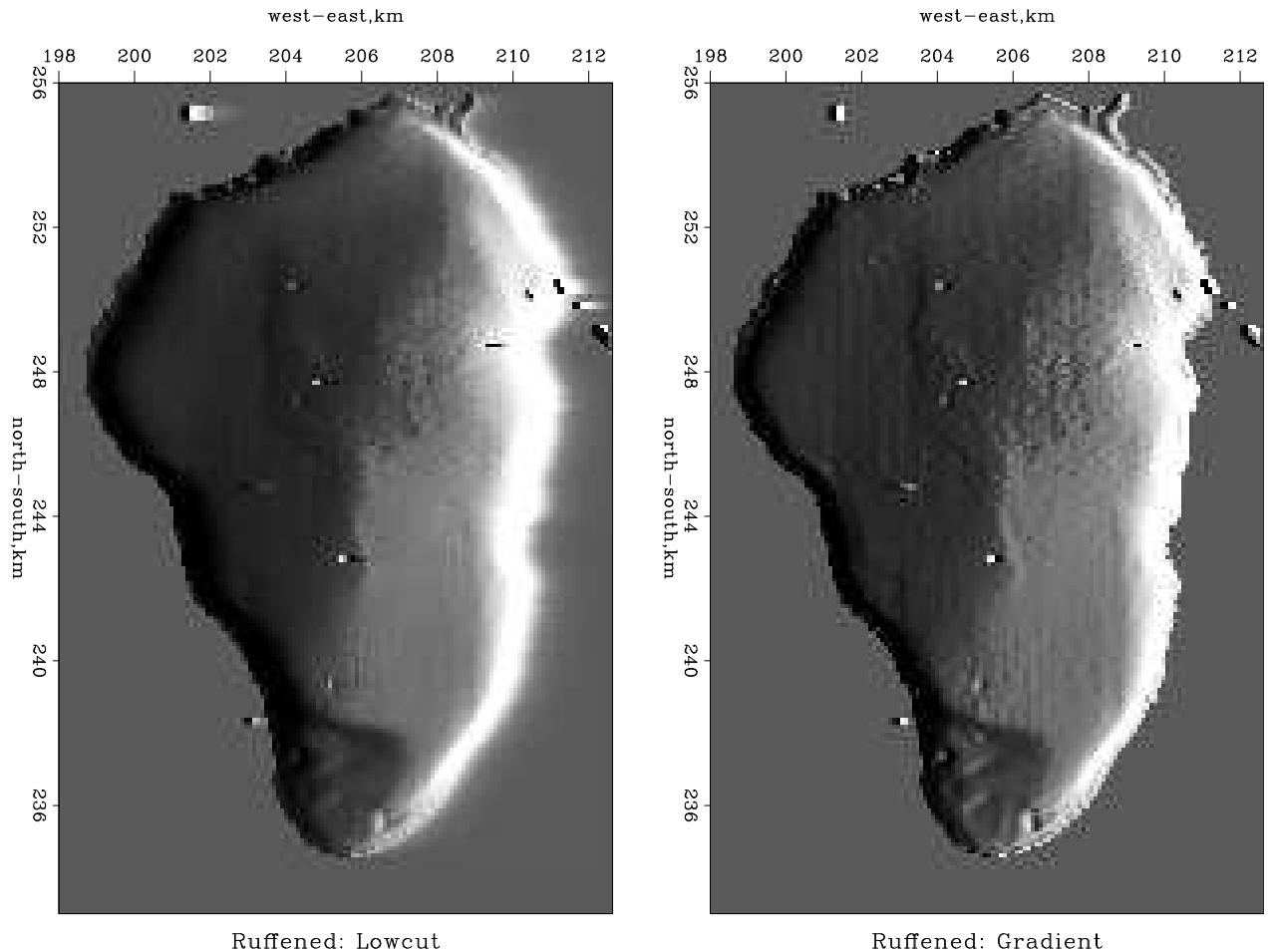


Figure 1.6: The depth of the Sea of Galilee after roughening. [ER,M]

ajt-galocut90

“**box car**”:

$$\frac{1-Z^5}{1-Z} \quad = \quad 1+Z+Z^2+Z^3+Z^4 \tag{1.27}$$

The filter (1.27) gives a moving average under a *rectangular* window. This is a basic smoothing filter. A clever way to apply it is to move the rectangle by adding a new value at one end while dropping an old value from the other end. This approach is formalized by the polynomial division algorithm, which can be simplified because so many coefficients are either one or zero. To find the recursion associated with $Y(Z) = X(Z)(1-Z^5)/(1-Z)$, we identify the coefficient of Z^t in $(1-Z)Y(Z) = X(Z)(1-Z^5)$. The result is

$$y_t \quad = \quad y_{t-1} + x_t - x_{t-5}. \tag{1.28}$$

This approach boils down to the program `boxconv()` which is so fast it is almost free! box_smooth Its last line scales the output by dividing by the rectangle length. With this scaling, the zero-frequency component of the input is unchanged, while other frequencies are suppressed.

```
module box_smooth {
  contains
  subroutine boxconv( nbox, nx, xx, yy) {
    integer,      intent(in)          ::nx,nbox
    integer                           ::i,ny
    real, dimension (:), intent (in) ::xx
    real, dimension (:), intent (out)::yy
    real, dimension (:), allocatable ::bb
    allocate(bb(nx+nbox))
    if( nbox < 1 || nbox > nx)  call erexit('boxconv')  # "||" means .OR.
    ny = nx+nbox-1
    bb(1) = xx(1)
    do i= 2, nx    { bb(i) = bb(i-1) + xx(i) } # B(Z) = X(Z)/(1-Z)
    do i= nx+1, ny { bb(i) = bb(i-1)         }
    do i= 1, nbox    { yy(i) = bb(i)         }
    do i= nbox+1, ny { yy(i) = bb(i) - bb(i-nbox)} # Y(Z) = B(Z)*(1-Z**nbox)
    do i= 1, ny    { yy(i) = yy(i) / nbox    }
    deallocate(bb)
  }
}
```

Back

1.1.14. Smoothing with a triangle

Triangle smoothing is rectangle smoothing done twice. For a mathematical description of the triangle filter, we simply square equation (1.27). Convolving a rectangle function with itself many times yields a result that mathematically tends towards a **Gaussian** function. Despite the sharp corner on the top of the triangle function, it has a shape that is remarkably similar to a Gaussian. Convolve a triangle with itself and you will see a very nice approximation to a Gaussian (the central limit theorem).

With filtering, **end effect**s can be a nuisance. Filtering increases the length of the data, but people generally want to keep input and output the same length (for various practical reasons). This is particularly true when filtering a space axis. Suppose the five-point signal $(1,1,1,1,1)$ is smoothed using the `boxconv()` program with the three-point smoothing filter $(1,1,1)/3$. The output is $5+3-1$ points long, namely, $(1,2,3,3,3,2,1)/3$. We could simply abandon the points off the ends, but I like to **fold** them back in, getting instead $(1+2,3,3,3,1+2)$. An advantage of the folding is that a constant-valued signal is unchanged by the smoothing. This is desirable since a smoothing filter is a low-pass filter which naturally should pass the lowest frequency $\omega = 0$ without distortion. The result is like a wave reflected by

a **zero-slope** end condition. Impulses are smoothed into triangles except near the boundaries. What happens near the boundaries is shown in Figure 1.7. Note that at

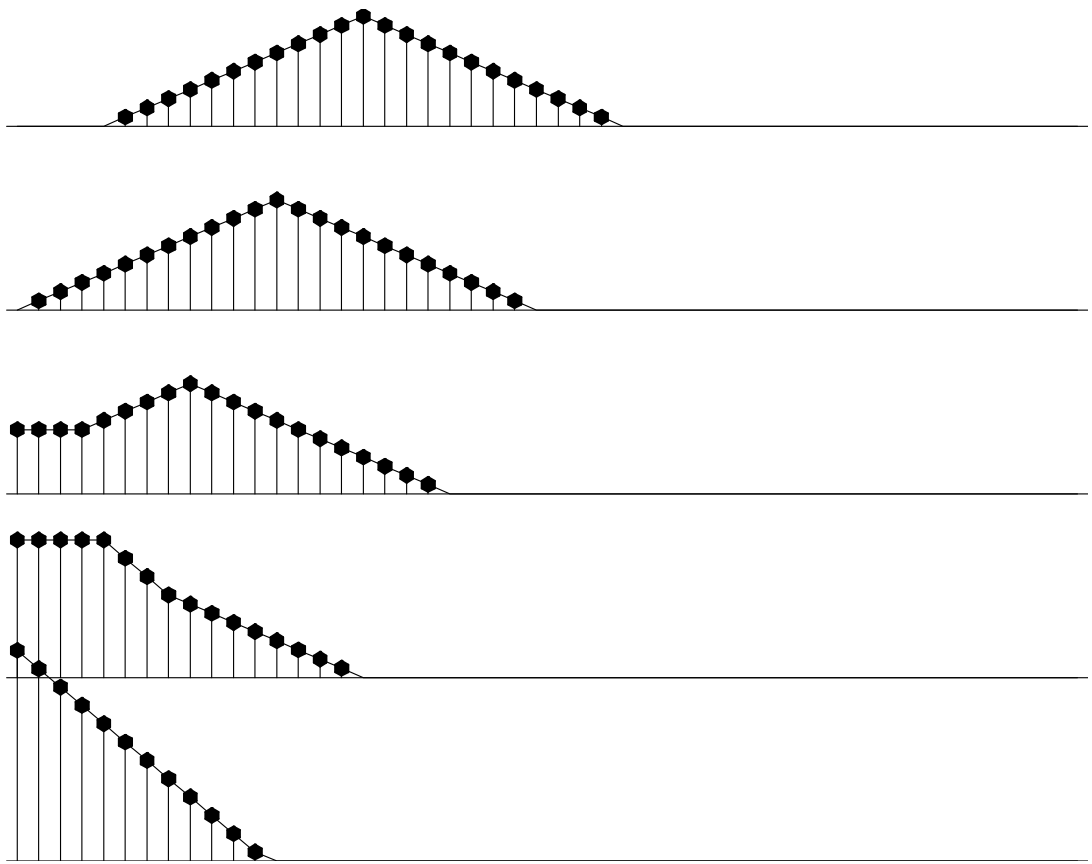


Figure 1.7: Edge effects when smoothing an impulse with a triangle function. Inputs are spikes at various distances from the edge. ajt-triend [ER,M]

the boundary, there is necessarily only half a triangle, but it is twice as tall.

```
  module triangle_smooth { # Convolve with triangle
    use box_smooth
    contains
    subroutine triangle( nbox, nd, xx, yy) {
      integer,        intent(in)       ::nbox,nd
      integer                          ::i,np,nq
      real, dimension (:), intent (in) ::xx
      real, dimension (:), intent (out)::yy
      real, dimension (:), allocatable ::pp,qq
      allocate(pp(nd+nbox-1), qq(nd+nbox+nbox-2))
      call boxconv( nbox, nd, xx, pp);    np = nbox+nd-1
      call boxconv( nbox, np, pp, qq);    nq = nbox+np-1
      do i=1,nd     { yy(i)     =             qq(i+nbox-1)     }
      do i=1,nbox-1 { yy(i)     =yy(i)      + qq(nbox-
i  )     } # fold back near end
      do i=1,nbox-1 { yy(nd-i+1)=yy(nd-i+1) + qq(nd+(nbox-
1)+i)} # fold back far end
      deallocate(pp,qq)
    }}
```

Back

Figure 1.7 was derived from the routine `triangle()`. triangle_smooth

1.1.15. Nearest-neighbor normal moveout (NMO)

Normal-moveout correction (**NMO**) is a geometrical correction of reflection seismic data that stretches the time axis so that data recorded at nonzero separation x_0 of shot and receiver, after stretching, appears to be at $x_0 = 0$. NMO correction is roughly like time-to-depth conversion with the equation $v^2 t^2 = z^2 + x_0^2$. After the data at x_0 is stretched from t to z, it should look like stretched data from any other x (assuming these are plane horizontal reflectors, etc.). In practice, z is not used; rather, **traveltime depth** τ is used, where $\tau = z/v$; so $t^2 = \tau^2 + x_0^2/v^2$. (Because of the limited alphabet of programming languages, I often use the keystroke z to denote τ.)

Typically, many receivers record each shot. Each seismogram can be transformed by NMO and the results all added. This is called "**stack**ing" or "**NMO stack**ing." The adjoint to this operation is to begin from a model which ideally is the zero-offset trace and spray this model to all offsets. From a matrix viewpoint, stacking is like a *row* vector of normal moveout operators and modeling is like a

column. An example is shown in Figure 1.8.

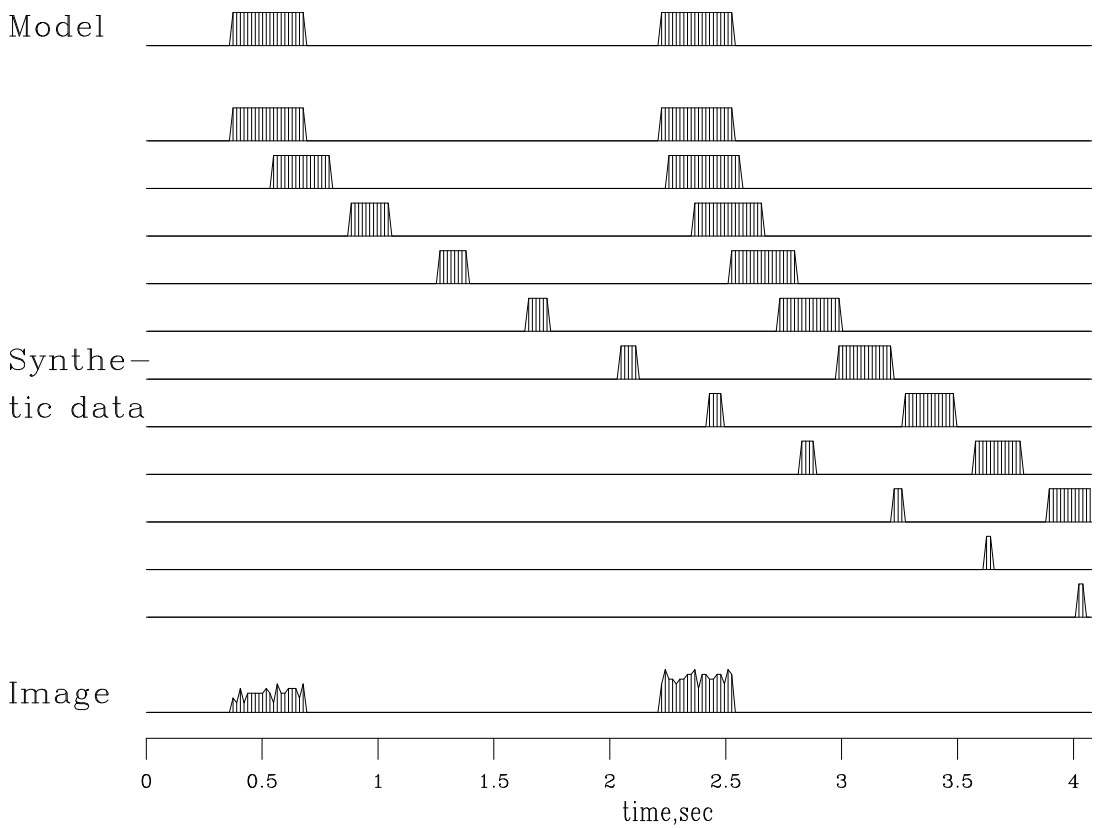


Figure 1.8: Hypothetical model, synthetic data, and model image. ajt-cunha [ER]

A module that does reverse moveout is `hypotenusei`. Given a zero-offset trace, it makes another at non-zero offset. The adjoint does the usual normal moveout

```
module hypotenusei {                                    # Inverse normal moveout
integer :: nt
integer, dimension (nt), allocatable :: iz
#%  _init( nt, t0, dt, xs)
integer it
real t0, dt, xs, t, zsquared
do it= 1, nt {  t = t0 + dt*(it-1)
        zsquared =  t * t - xs * xs
        if ( zsquared >= 0.)
                iz (it) = 1.5 + (sqrt( zsquared) - t0) /dt
        else
                iz (it) = 0
        }
#%  _lop(   zz, tt)
integer   it
do it= 1, nt {
             if ( iz (it) > 0 ) {
                             if( adj)   zz( iz(it))  +=  tt(    it )
                             else       tt(it)       +=  zz( iz(it))
                }
        }
}
```

Back

```
module imospray {                       # inverse moveout and spray into a gather.
use hypotenusei
real :: x0,dx, t0,dt
integer :: nx,nt
#%  _init (slow, x0,dx, t0,dt, nt,nx)
                real slow
                x0 = x0*slow
                dx = dx*slow
#% _lop(  stack(nt),    gather(nt,nx))
integer ix, stat
do ix= 1, nx {
        call hypotenusei_init ( nt, t0, dt, x0 + dx*(ix-1))
        stat = hypotenusei_lop ( adj, .true., stack, gather(:,ix))
        }
call hypotenusei_close ()
}
```

Back

correction. hypotenusei (My 1992 textbook (PVI) illustrates many additional features of normal moveout.) A companion routine `imospray` loops over offsets and makes a trace for each. The adjoint of `imospray` is the industrial process of moveout and stack. imospray

This is the first time we have built an operator (moveout and stack) from a collection of other operators (moveout at various offsets) and there are new possibilities for confusion and program bugs. As earlier with the matrix multiplication operator, to use the `imospray` operator, there are two steps, one that we use to set things up

```
call   imospray_init( slow, x0,dx, t0,dt, nt,nx)
```

and another step that we use a lot in the next chapter for analysis and data fitting.

```
stat = imospray_lop( adj, add, stack, gather)
```

Later we'll see programs that are not operators. Every program that is an operator (contains `#% _init` and `#% _lop`) is expanded by Loptran to a longer Fortran code where the `_lop` function begins by (potentially) erasing the output (when `add=.false.`). This potential erasing is done in both `hypotenusei` and `imospray`. Consider the the adjoint of spraying which is stacking. Here the occurance of the `add=.true.` in `imospray` assures we do not erase the stack each time we add in

another trace. Because of Loptran we don't explicitly see that `imospray_lop` has its own potential erase of its output which we'd turn off if we wanted to add one stack upon another.

1.1.16. Coding chains and arrays

With a collection of operators, we can build more elaborate operators. One method is chaining. For example, the operator product $\mathbf{A} = \mathbf{BC}$ is represented in the subroutine `chain2( op1, op2, ...)`. Likewise the operator product $\mathbf{A} = \mathbf{BCD}$ is represented in the in the subroutine `chain3( op1, op2, op3,...)`. Another way to make more elaborate operators is to put operators in a matrix such as subroutine `array` also in module `smallchain2`. smallchain2

1.2. ADJOINT DEFINED: DOT-PRODUCT TEST

Having seen many examples of **space**s, operators, and adjoints, we should now see more formal definitions because abstraction helps us push these concepts to their limits.

```
module chain1_mod {
  logical, parameter, private :: T = .true., F = .false.
  interface chain0
     module procedure chain20
     module procedure chain30
  end interface
contains
  subroutine row0(op1,op2, adj,add, m1, m2, d, eps) { #        ROW  d = Am1+epsBm2
    interface {
       integer function op1(adj,add,m,d){real::m(:),d(:);logical,intent(in)::adj,add}
       integer function op2(adj,add,m,d){real::m(:),d(:);logical,intent(in)::adj,add}
    }
    logical, intent (in) :: adj, add
    real, dimension (:)  :: m1,m2,d
    real                 :: eps
    integer              :: st
    if (adj) { st = op1 (T, add, m1, d)        # m1 = A'd
               st = op2 (T, add, m2, d)        # m2 = B'd
               m2 = eps*m2                     # m2 = eps(B'd)
    }
    else {     st = op2 (F, add, eps*m2, d)    # d = epsBm2
               st = op1 (F,   T,     m1, d)    # d = Am1+epsBm2
    }
  }
  subroutine chain20(op1,op2, adj,add, m,d,t1) { #        CHAIN 2
    interface {
       integer function op1(adj,add,m,d){real::m(:),d(:);logical,intent(in)::adj,add}
       integer function op2(adj,add,m,d){real::m(:),d(:);logical,intent(in)::adj,add}
    }
    logical, intent(in) :: adj, add
    real, dimension(:)  :: m,d, t1
    integer             :: st
    if(adj) {  st = op1( T,   F, t1,d )         #        t =    A' d
               st = op2( T, add, m, t1)         # m = B' t = B' A' d
    }
    else    {  st = op2( F,   F, m, t1)         #        t =    B  m
               st = op1( F, add, t1,d )         # d = A  t = A  B  m
    }
  }
```

1.2.1. Definition of a vector space

An operator transforms a **space** to another space. Examples of spaces are model space **m** and data space **d**. We think of these spaces as vectors whose components are packed with numbers, either real or complex numbers. The important practical concept is that not only does this packing include one-dimensional spaces like signals, two-dimensional spaces like images, 3-D movie cubes, and zero-dimensional spaces like a data mean, etc, but spaces can be sets of all the above. One space that is a set of three cubes is the earth's magnetic field, which has three components; and each component is a function of a three-dimensional space. (The 3-D *physical space* we live in is not the abstract ***vector space*** of models and data so abundant in this book. Here the word "space" without an adjective means the vector space.)

A more heterogeneous example of a vector space is **data tracks**. A depth-sounding survey of a lake can make a vector space that is a collection of tracks, a vector of vectors (each vector having a different number of components, because lakes are not square). This vector space of depths along tracks in a lake contains the depth values only. The (x, y)-coordinate information locating each measured depth value is (normally) something outside the vector space. A data space could also be a collection of echo soundings, waveforms recorded along tracks.

We briefly recall information about vector spaces found in elementary books: Let α be any scalar. Then if $\mathbf{d}_1$ is a vector and $\mathbf{d}_2$ is conformable with it, then other vectors are $\alpha\mathbf{d}_1$ and $\mathbf{d}_1+\mathbf{d}_2$. The size measure of a vector is a positive value called a norm. The norm is usually defined to be the **dot product** (also called the L_2 **norm**), say $\mathbf{d}\cdot\mathbf{d}$. For complex data it is $\mathbf{d}\cdot\bar{\mathbf{d}}$ where $\bar{\mathbf{d}}$ is the complex conjugate of $\mathbf{d}$. In theoretical work the "length of a vector" means the vector's norm. In computational work the "length of a vector" means the number of components in the vector.

Norms generally include a **weighting function**. In physics, the norm generally measures a conserved quantity like energy or momentum, so, for example, a weighting function for magnetic flux is permittivity. In data analysis, the proper choice of the weighting function is a practical statistical issue, discussed repeatedly throughout this book. The algebraic view of a weighting function is that it is a diagonal matrix with positive values $w(i)\geq 0$ spread along the diagonal, and it is denoted $\mathbf{W}=\mathbf{diag}[w(i)]$. With this weighting function the L_2 norm of a data space is denoted $\mathbf{d}'\mathbf{W}\mathbf{d}$. Standard notation for norms uses a double absolute value, where $||\mathbf{d}||=\mathbf{d}'\mathbf{W}\mathbf{d}$. A central concept with norms is the triangle inequality, $||\mathbf{d}_1+\mathbf{d}_2||\leq||\mathbf{d}_1||+||\mathbf{d}_2||$ whose proof you might recall (or reproduce with the use of dot products).

1.2.2. Dot-product test for validity of an adjoint

There is a huge gap between the conception of an idea and putting it into practice. During development, things fail far more often than not. Often, when something fails, many tests are needed to track down the cause of failure. Maybe the cause cannot even be found. More insidiously, failure may be below the threshold of detection and poor performance suffered for years. The **dot-product test** enables us to ascertain whether the program for the adjoint of an operator is precisely consistent with the operator itself. It can be, and it should be.

Conceptually, the idea of matrix transposition is simply $a'_{ij} = a_{ji}$. In practice, however, we often encounter matrices far too large to fit in the memory of any computer. Sometimes it is also not obvious how to formulate the process at hand as a matrix multiplication. (Examples are differential equations and fast Fourier transforms.) What we find in practice is that an application and its adjoint amounts to two routines. The first routine amounts to the matrix multiplication $\mathbf{Fx}$. The adjoint routine computes $\mathbf{F}'\mathbf{y}$, where $\mathbf{F}'$ is the **conjugate-transpose** matrix. In later chapters we will be solving huge sets of simultaneous equations, in which both routines are required. If the pair of routines are inconsistent, we are doomed from the start. The dot-product test is a simple test for verifying that the two routines are

adjoint to each other.

The associative property of linear algebra says that we do not need parentheses in a vector-matrix-vector product like $\mathbf{y}'\mathbf{F}\mathbf{x}$ because we get the same result no matter where we put the parentheses. They serve only to determine the sequence of computation. Thus,

$$\mathbf{y}'(\mathbf{F}\mathbf{x}) \quad = \quad (\mathbf{y}'\mathbf{F})\mathbf{x} \tag{1.29}$$

$$\mathbf{y}'(\mathbf{F}\mathbf{x}) \quad = \quad (\mathbf{F}'\mathbf{y})'\mathbf{x} \tag{1.30}$$

(In general, the matrix is not square.) To perform the dot-product test, load the vectors $\mathbf{x}$ and $\mathbf{y}$ with random numbers. Using your program for $\mathbf{F}$, compute the vector $\tilde{\mathbf{y}} = \mathbf{F}\mathbf{x}$, and using your program for $\mathbf{F}'$, compute $\tilde{\mathbf{x}} = \mathbf{F}'\mathbf{y}$. Inserting these into equation (1.30) gives you two scalars that should be equal.

$$\mathbf{y}'(\mathbf{F}\mathbf{x}) \quad = \quad \mathbf{y}'\tilde{\mathbf{y}} \quad = \quad \tilde{\mathbf{x}}'\mathbf{x} \quad = \quad (\mathbf{F}'\mathbf{y})'\mathbf{x} \tag{1.31}$$

The left and right sides of this equation will be computationally equal only if the program doing $\mathbf{F}'$ is indeed adjoint to the program doing $\mathbf{F}$ (unless the random numbers do something miraculous). A final word: If an operator uses complex arithmetic then both its input and output are "in the field of complex numbers." The conver-

sion between real and complex numbers is not a linear operator despite its seeming similarity to truncation and zero padding.

The program for applying the dot product test is `dot_test` /prog:dottest. The Fortran way of passing a linear operator as an argument is to specify the function interface. Fortunately, we have already defined the interface for a generic linear operator. To use the `dot_test` program, you need to initialize an operator with specific arguments (the `_init` subroutine) and then pass the operator itself (the `_lop` function) to the test program. You also need to specify the sizes of the model and data vectors so that temporary arrays can be constructed. The program runs the dot product test twice, second time with `add = .true.` to test if the operator can be used properly for accumulating the result like $\mathbf{y} \leftarrow \mathbf{y} + \mathbf{B}\mathbf{x}$. dottest

I tested (1.31) on many operators and was surprised and delighted to find that it is often satisfied to an accuracy near the computing precision. I do not doubt that larger rounding errors could occur, but so far, every time I encountered a relative discrepancy of 10^{-5} or more, I was later able to uncover a conceptual or programming error. Naturally, when I do dot-product tests, I scale the implied matrix to a small dimension in order to speed things along, and to be sure that boundaries are not overwhelmed by the much larger interior.

```
module dottest {
  logical, parameter, private ::  T = .true.,  F = .false.
contains
  subroutine dot_test( oper, n_mod, n_dat, dot1, dot2) {
    integer,              intent (in)  :: n_mod, n_dat
    real, dimension (2), intent (out) :: dot1, dot2
    interface {
       function oper( adj, add, mod, dat) result(stat) {
            integer                :: stat
            logical, intent (in) :: adj, add
            real, dimension (:)  :: mod, dat
            }
       }
    real, dimension( n_mod)           :: mod1, mod2
    real, dimension( n_dat)           :: dat1, dat2
    integer                           :: stat
    call random_number( mod1);  call random_number( dat2)
    stat = oper( F, F, mod1, dat1);  dot1( 1) = dot_product( dat1, dat2)
    stat = oper( T, F, mod2, dat2);  dot1( 2) = dot_product( mod1, mod2)
    write (0,*) dot1
    stat = oper( F, T, mod1, dat1);  dot2( 1) = dot_product( dat1, dat2)
    stat = oper( T, T, mod2, dat2);  dot2( 2) = dot_product( mod1, mod2)
    write (0,*) dot2
  }
}
```

Back

Do not be alarmed if the operator you have defined has **truncation** errors. Such errors in the definition of the original operator should be identically matched by truncation errors in the adjoint operator. If your code passes the **dot-product test**, then you really have coded the adjoint operator. In that case, to obtain inverse operators, you can take advantage of the standard methods of mathematics.

We can speak of a **continuous function** $f(t)$ or a **discrete function** f_t. For continuous functions we use integration, and for discrete ones we use summation. In formal mathematics, the dot-product test *defines* the adjoint operator, except that the summation in the dot product may need to be changed to an integral. The input or the output or both can be given either on a continuum or in a discrete domain. So the dot-product test $\mathbf{y}'\tilde{\mathbf{y}} = \tilde{\mathbf{x}}'\mathbf{x}$ could have an integration on one side of the equal sign and a summation on the other. Linear-operator theory is rich with concepts not developed here.

1.2.3. The word "adjoint"

In mathematics the word "**adjoint**" has three meanings. One of them, the so-called **Hilbert adjoint**, is the one generally found in physics and engineering and it is the

one used in this book. In linear algebra is a different matrix, called the **adjugate** matrix. It is a matrix whose elements are signed cofactors (minor determinants). For invertible matrices, this matrix is the **determinant** times the **inverse matrix**. It can be computed without ever using division, so potentially the adjugate can be useful in applications where an inverse matrix does not exist. Unfortunately, the adjugate matrix is sometimes called the adjoint matrix, particularly in the older literature. Because of the confusion of multiple meanings of the word adjoint, in the first printing of PVI, I avoided the use of the word and substituted the definition, "**conjugate transpose**". Unfortunately this was often abbreviated to "conjugate," which caused even more confusion. Thus I decided to use the word adjoint and have it always mean the Hilbert adjoint found in physics and engineering.

1.2.4. Matrix versus operator

Here is a short summary of where we have been and where we are going: Start from the class of linear operators, add subscripts and you get matrices. Examples of operators without subscripts are routines that solve differential equations and routines that do fast Fourier transform. What people call "sparse matrices" are often

not really matrices but operators, because they are not defined by data structures but by routines that apply them to a vector. With sparse matrices you easily can do $\mathbf{A}(\mathbf{B}(\mathbf{C}\mathbf{x}))$ but not $(\mathbf{ABC})\mathbf{x}$.

Although a linear operator does not have defined subscripts, you can determine what would be the operator value at any subscript: by applying the operator to an impulse function, you would get a matrix column. The adjoint operator is one from which we can extract the transpose matrix. For large spaces this extraction is unwieldy, so to test the validity of adjoints, we probe them with random vectors, say $\mathbf{x}$ and $\mathbf{y}$, to see whether $\mathbf{y}'(\mathbf{A}\mathbf{x}) = (\mathbf{A}'\mathbf{y})'\mathbf{x}$. Mathematicians define adjoints by this test, except that instead of using random vectors, they say "for all functions," which includes the continuum.

This defining test makes adjoints look mysterious. Careful inspection of operator adjoints, however, generally reveals that they are built up from simple matrices. Given adjoints $\mathbf{A}'$, $\mathbf{B}'$, and $\mathbf{C}'$, the adjoint of $\mathbf{ABC}$ is $\mathbf{C}'\mathbf{B}'\mathbf{A}'$. Fourier transforms and linear-differential-equation solvers are chains of matrices, so their adjoints can be assembled by the application of adjoint components in reverse order. The other way we often see complicated operators being built from simple ones is when operators are put into components of matrices, typically a 1×2 or 2×1 matrix containing

two operators. An example of the adjoint of a two-component column operator is

$$\begin{bmatrix} \mathbf{A} \\ \mathbf{B} \end{bmatrix}' \quad = \quad \begin{bmatrix} \mathbf{A}' & \mathbf{B}' \end{bmatrix} \tag{1.32}$$

Although in practice an operator might be built from matrices, fundamentally, a matrix is a data structure whereas an operator is a procedure. A matrix is an operator if its subscripts are hidden but it can be applied to a space, producing another space.

As matrices have inverses, so do linear operators. You don't need subscripts to find an inverse. The conjugate-gradient method and conjugate-direction method explained in the next chapter are attractive methods of finding them. They merely apply $\mathbf{A}$ and $\mathbf{A}'$ and use inner products to find coefficients of a polynomial in $\mathbf{A}\mathbf{A}'$ that represents the inverse operator.

Whenever we encounter a **positive-definite** matrix we should recognize its likely origin in a nonsymmetric matrix $\mathbf{F}$ times its adjoint. Those in natural sciences often work on solving simultaneous equations but fail to realize that they should return to the origin of the equations which is often a fitting goal; i.e., applying an operator to a model should yield data, i.e., $\mathbf{d} \approx \mathbf{d}_0 + \mathbf{F}(\mathbf{m} - \mathbf{m}_0)$ where the operator $\mathbf{F}$ is a partial derivative matrix (and there are potential underlying nonlinearities).

This begins another story with new ingredients, weighting functions and statistics.

1.2.5. Inverse operator

A common practical task is to fit a vector of observed data $\mathbf{d}_{\text{obs}}$ to some theoretical data $\mathbf{d}_{\text{theor}}$ by the adjustment of components in a vector of model parameters $\mathbf{m}$.

$$\mathbf{d}_{\text{obs}} \quad \approx \quad \mathbf{d}_{\text{theor}} \quad = \quad \mathbf{F}\mathbf{m} \tag{1.33}$$

A huge volume of literature establishes theory for two estimates of the model, $\hat{\mathbf{m}}_1$ and $\hat{\mathbf{m}}_2$, where

$$\hat{\mathbf{m}}_1 \quad = \quad (\mathbf{F}'\mathbf{F})^{-1}\mathbf{F}'\mathbf{d} \tag{1.34}$$

$$\hat{\mathbf{m}}_2 \quad = \quad \mathbf{F}'(\mathbf{F}\mathbf{F}')^{-1}\mathbf{d} \tag{1.35}$$

Some reasons for the literature being huge are the many questions about the existence, quality, and cost of the inverse operators. Before summarizing that, let us quickly see why these two solutions are reasonable. Inserting equation (1.33) into equation (1.34), and inserting equation (1.35) into equation (1.33), we get the rea-

sonable statements:

$$\hat{\mathbf{m}}_1 \quad = \quad (\mathbf{F}'\mathbf{F})^{-1}(\mathbf{F}'\mathbf{F})\mathbf{m} \quad = \quad \mathbf{m} \tag{1.36}$$

$$\hat{\mathbf{d}}_{\text{theor}} \quad = \quad (\mathbf{F}\mathbf{F}')(\mathbf{F}\mathbf{F}')^{-1}\mathbf{d} \quad = \quad \mathbf{d} \tag{1.37}$$

Equation (1.36) says that the estimate $\hat{\mathbf{m}}_1$ gives the correct model $\mathbf{m}$ if you start from the theoretical data. Equation (1.37) says that the model estimate $\hat{\mathbf{m}}_2$ gives the theoretical data if we derive $\hat{\mathbf{m}}_2$ from the theoretical data. Both of these statements are delightful. Now let us return to the problem of the inverse matrices.

Strictly speaking, a rectangular matrix does not have an inverse. Surprising things often happen, but commonly, when $\mathbf{F}$ is a tall matrix (more data values than model values) then the matrix for finding $\hat{\mathbf{m}}_1$ is invertible while that for finding $\hat{\mathbf{m}}_2$ is not, and when the matrix is wide instead of tall (the number of data values is less than the number of model values) it is the other way around. In many applications neither $\mathbf{F}'\mathbf{F}$ nor $\mathbf{F}\mathbf{F}'$ is invertible. This difficulty is solved by "**damping**" as we will see in later chapters. The point to notice in this chapter on adjoints is that in any application where $\mathbf{F}\mathbf{F}'$ or $\mathbf{F}'\mathbf{F}$ equals $\mathbf{I}$ (unitary operator), that the adjoint operator $\mathbf{F}'$ is the inverse $\mathbf{F}^{-1}$ by either equation (1.34) or (1.35).

Theoreticians like to study inverse problems where $\mathbf{m}$ is drawn from the field of

continuous functions. This is like the vector **m** having infinitely many components. Such problems are hopelessly intractable unless we find, or assume, that the operator $\mathbf{F}'\mathbf{F}$ is an identity or diagonal matrix.

In practice, theoretical considerations may have little bearing on how we proceed. Current computational power limits matrix inversion jobs to about 10^4 variables. This book specializes in big problems, those with more than about 10^4 variables, but the methods we learn are also excellent for smaller problems.

1.2.6. Automatic adjoints

Computers are not only able to perform computations; they can do mathematics. Well known software is Mathematica and Maple. Adjoints can also be done by symbol manipulation. For example Ralf Giering[4] offers a program for converting linear operator programs into their adjoints.

---

[4] http://klima47.dkrz.de/giering/tamc/tamc.html

EXERCISES:

1 Suppose a linear operator $\mathbf{F}$ has its input in the discrete domain and its output in the continuum. How does the operator resemble a matrix? Describe the operator $\mathbf{F}'$ that has its input in the discrete domain and its output in the continuum. To which do you apply the words "scales and adds some functions," and to which do you apply the words "does a bunch of integrals"? What are the integrands?

Chapter 2

Model fitting by least squares

The first level of computer use in science and engineering is **modeling**. Beginning from physical principles and design ideas, the computer mimics nature. After this, the worker looks at the result and thinks a while, then alters the modeling program

and tries again. The next, deeper level of computer use is that the computer itself examines the results of modeling and reruns the modeling job. This deeper level is variously called "**fitting**" or "**estimation**" or "**inversion**." We inspect the **conjugate-direction method** of fitting and write a subroutine for it that will be used in most of the examples in this monograph.

2.1. HOW TO DIVIDE NOISY SIGNALS

If "inversion" is dividing by a matrix, then the place to begin is dividing one number by another, say one function of frequency by another function of frequency. A single parameter fitting problem arises in Fourier analysis, where we seek a "best answer" at each frequency, then combine all the frequencies to get a best signal. Thus emerges a wide family of interesting and useful applications. However, Fourier analysis first requires us to introduce complex numbers into statistical estimation.

Multiplication in the Fourier domain is **convolution** in the time domain. Fourier-domain division is time-domain **deconvolution**. This division is challenging when the divisor has observational error. Failure erupts if zero division occurs. More insidious are the poor results we obtain when zero division is avoided by a near

miss.

2.1.1. Dividing by zero smoothly

Think of any real numbers x, y, and f where $y = xf$. Given y and f we see a computer program containing $x = y/f$. How can we change the program so that it never divides by zero? A popular answer is to change $x = y/f$ to $x = yf/(f^2 + \epsilon^2)$, where ϵ is any tiny value. When $|f| >> |\epsilon|$, then x is approximately y/f as expected. But when the divisor f vanishes, the result is safely zero instead of infinity. The transition is smooth, but some criterion is needed to choose the value of ϵ. This method may not be the only way or the best way to cope with **zero division**, but it is a good way, and it permeates the subject of signal analysis.

To apply this method in the Fourier domain, suppose that X, Y, and F are complex numbers. What do we do then with $X = Y/F$? We multiply the top and bottom by the complex conjugate $\overline{F}$, and again add ϵ^2 to the denominator. Thus,

$$X(\omega) \quad = \quad \frac{\overline{F(\omega)}\, Y(\omega)}{\overline{F(\omega)} F(\omega) + \epsilon^2} \tag{2.1}$$

Now the denominator must always be a positive number greater than zero, so division is always safe. Equation (2.1) ranges continuously from **inverse filter**ing, with $X = Y/F$, to filtering with $X = \overline{F}Y$, which is called "**matched filter**ing." Notice that for any complex number F, the phase of $1/F$ equals the phase of $\overline{F}$, so the filters $1/F$ and $\overline{F}$ have inverse amplitudes but identical phase.

2.1.2. Damped solution

Equation (2.1) is the solution to an optimization problem that arises in many applications. Now that we know the solution, let us formally define the problem. First, we will solve a simpler problem with real values: we will choose to minimize the **quadratic function** of x:

$$Q(x) \quad = \quad (fx - y)^2 + \epsilon^2 x^2 \tag{2.2}$$

The second term is called a "**damping** factor" because it prevents x from going to $\pm\infty$ when $f \to 0$. Set $dQ/dx = 0$, which gives

$$0 \quad = \quad f(fx - y) + \epsilon^2 x \tag{2.3}$$

This yields the earlier answer $x = fy/(f^2 + \epsilon^2)$.

With Fourier transforms, the signal X is a complex number at each frequency ω. So we generalize equation (2.2) to

$$Q(\bar{X},X) \quad = \quad (\overline{FX-Y})(FX-Y)+\epsilon^2\bar{X}X \quad = \quad (\bar{X}\bar{F}-\bar{Y})(FX-Y)+\epsilon^2\bar{X}X \tag{2.4}$$

To minimize Q we could use a real-values approach, where we express $X = u + iv$ in terms of two real values u and v and then set $\partial Q/\partial u = 0$ and $\partial Q/\partial v = 0$. The approach we will take, however, is to use complex values, where we set $\partial Q/\partial X = 0$ and $\partial Q/\partial \bar{X} = 0$. Let us examine $\partial Q/\partial \bar{X}$:

$$\frac{\partial Q(\bar{X},X)}{\partial \bar{X}} \quad = \quad \bar{F}(FX-Y)+\epsilon^2 X \quad = \quad 0 \tag{2.5}$$

The derivative $\partial Q/\partial X$ is the complex conjugate of $\partial Q/\partial \bar{X}$. So if either is zero, the other is too. Thus we do not need to specify both $\partial Q/\partial X = 0$ and $\partial Q/\partial \bar{X} = 0$. I usually set $\partial Q/\partial \bar{X}$ equal to zero. Solving equation (2.5) for X gives equation (2.1).

Equation (2.1) solves $Y = XF$ for X, giving the solution for what is called "the **deconvolution** problem with a known wavelet F." Analogously we can use $Y = XF$ when the filter F is unknown, but the input X and output Y are given. Simply interchange X and F in the derivation and result.

2.1.3. Smoothing the denominator spectrum

Equation (2.1) gives us one way to divide by zero. Another way is stated by the equation

$$X(\omega) \quad = \quad \frac{\overline{F(\omega)}\, Y(\omega)}{\langle \overline{F(\omega)} F(\omega) \rangle} \tag{2.6}$$

where the strange notation in the denominator means that the spectrum there should be smoothed a little. Such smoothing fills in the holes in the spectrum where zero-division is a danger, filling not with an arbitrary numerical value ϵ but with an average of nearby spectral values. Additionally, if the denominator spectrum $\overline{F(\omega)}F(\omega)$ is rough, the smoothing creates a shorter autocorrelation function.

Both divisions, equation (2.1) and equation (2.6), irritate us by requiring us to specify a parameter, but for the latter, the parameter has a clear meaning. In the latter case we smooth a spectrum with a smoothing window of width, say $\Delta\omega$ which this corresponds inversely to a time interval over which we smooth. Choosing a numerical value for ϵ has not such a simple interpretation.

We jump from simple mathematical theorizing towards a genuine practical application when I grab some real data, a function of time and space from another

textbook. Let us call this data $f(t,x)$ and its 2-D Fourier transform $F(\omega,k_x)$. The data and its autocorrelation are in Figure 2.1.

The autocorrelation $a(t,x)$ of $f(t,x)$ is the inverse 2-D Fourier Transform of $\overline{F(\omega,k_x)}F(\omega,k_x)$. Autocorrelations $a(x,y)$ satisfy the symmetry relation $a(x,y) = a(-x,-y)$. Figure 2.2 shows only the interesting quadrant of the two independent quadrants. We see the autocorrelation of a 2-D function has some resemblance to the function itself but differs in important ways.

Instead of messing with two different functions X and Y to divide, let us divide F by itself. This sounds like $1 = F/F$ but we will watch what happens when we do the division carefully avoiding zero division in the ways we usually do.

Figure 2.2 shows what happens with

$$1 \quad = \quad F/F \quad \approx \quad \frac{\overline{F}F}{\overline{F}F + \epsilon^2} \tag{2.7}$$

and with

$$1 \quad = \quad F/F \quad \approx \quad \frac{\overline{F}F}{\langle \overline{F}F \rangle} \tag{2.8}$$

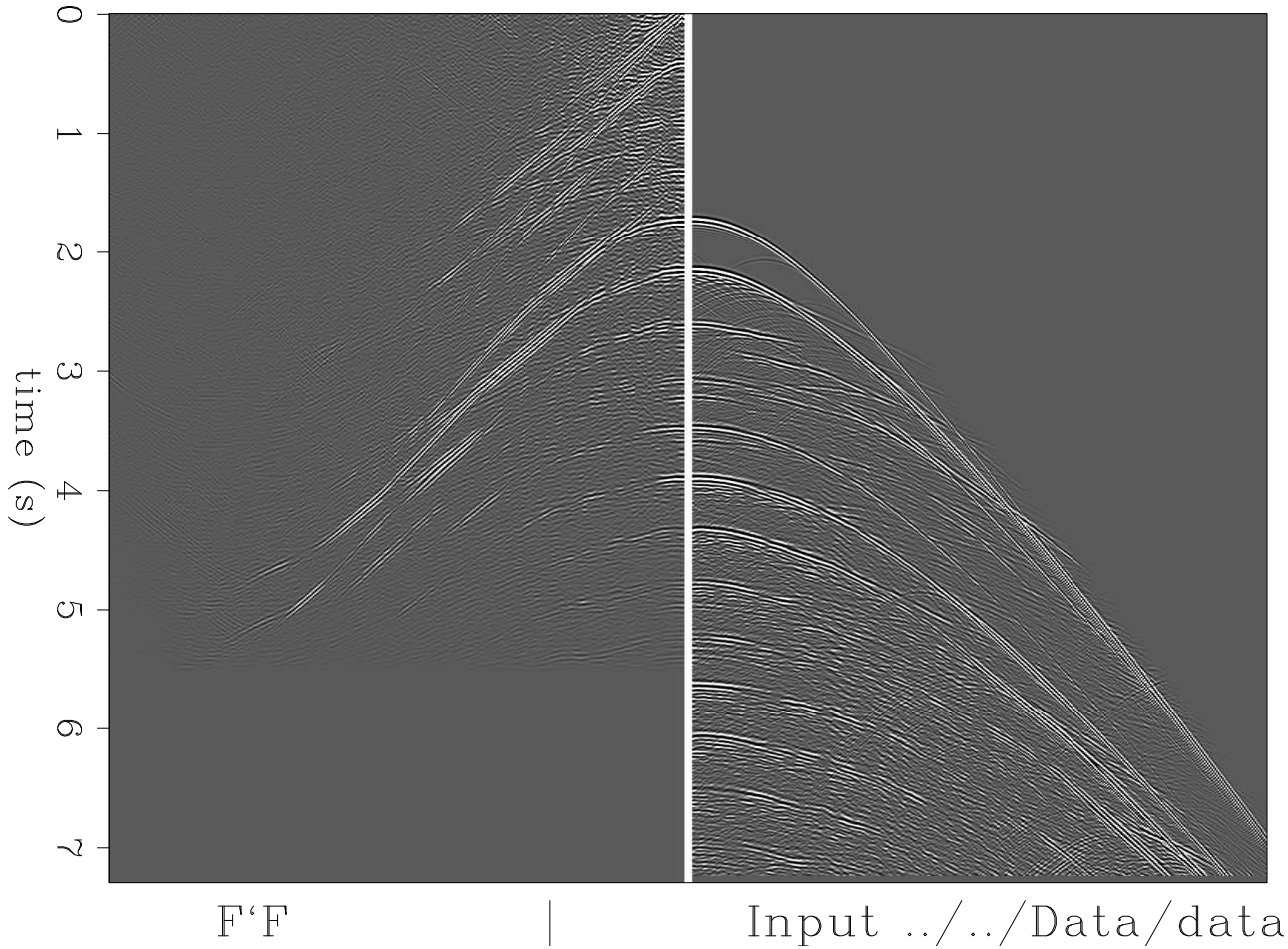


Figure 2.1: 2-D data (right) and a quadrant of its autocorrelation (left). Notice the longest nonzero time lag on the data is about 5.5 sec which is the latest nonzero signal on the autocorrelation. lsq-antoine10 [ER]

From Figure 2.2 we notice that both methods of avoiding zero division give similar results. By playing with the ϵ and the smoothing width the pictures could be made even more similar. My preference, however, is the smoothing. It is difficult to make physical sense of choosing a numerical value for ϵ. It is much easier to make physical sense of choosing a smoothing window. The smoothing window is in (ω, k_x) space, but Fourier transformation tells us its effect in (t, x) space.

2.1.4. Imaging

The example of dividing a function by itself ($1 = F/F$) might not seem to make much sense, but it is very closely related to estimation often encounted in imaging applications. It's not my purpose here to give a lecture on imaging theory, but here is an overbrief explanation.

Imagine a downgoing wavefield $D(\omega, x, z)$. Propagating against irregularities in the medium $D(\omega, x, z)$ creates by scattering an upgoing wavefield $U(\omega, x, z)$. Given U and D, if there is a stong temporal correlation between them at any (x, z) it likely means there is a reflector nearby that is manufacturing U from D. This reflectivity could be quantified by U/D. At the earth's surface the surface boundary condition

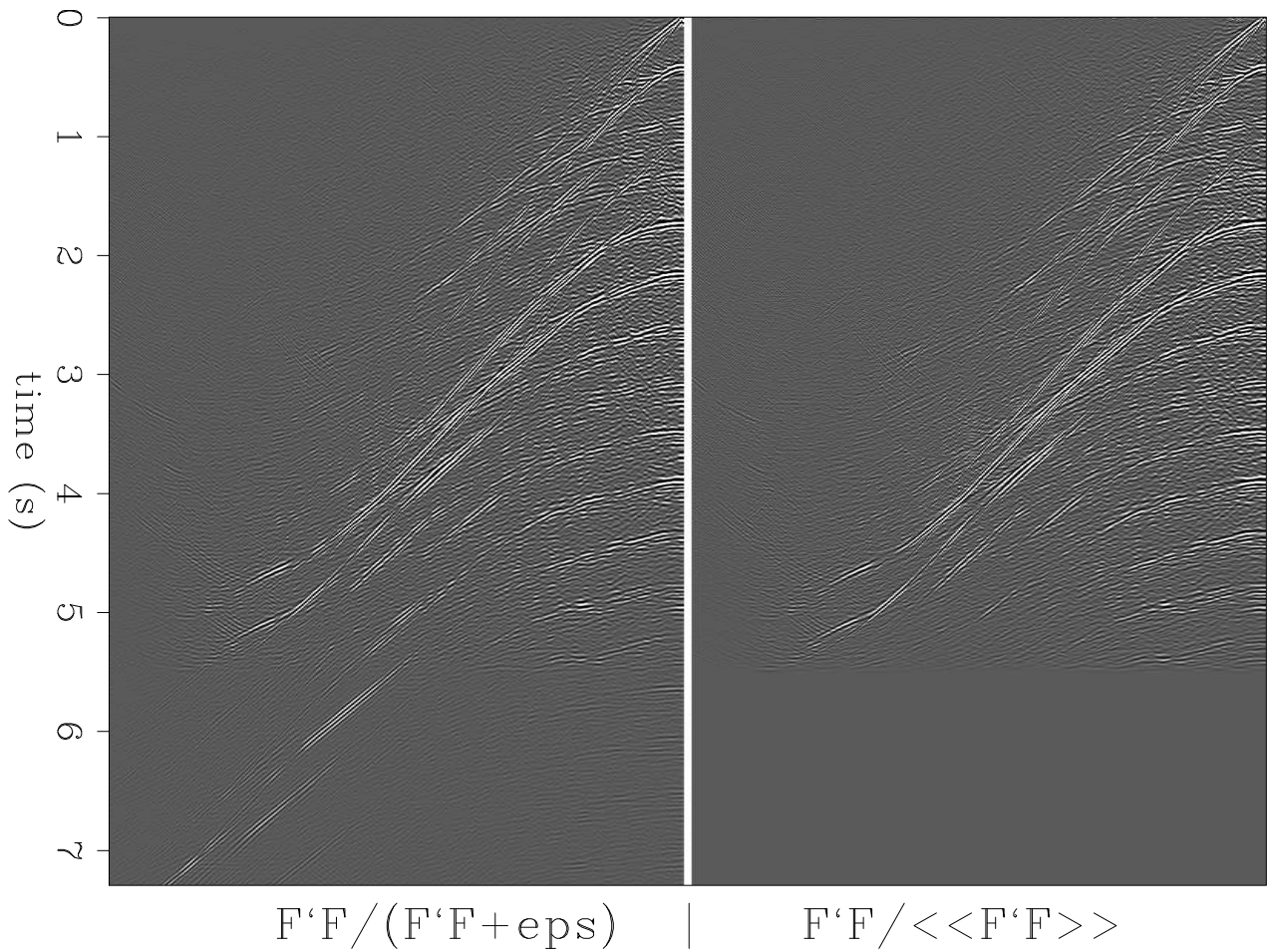


Figure 2.2: Equation (2.7) (left) and equation (2.8) (right). Both ways of dividing by zero give similar results. lsq-antoine11 [ER]

says something like $U = D$ or $U = -D$. Thus at the surface we have something like F/F. As we go down in the earth, the main difference is that U and D get time shifted in opposite directions, so U and D are similar but for that time difference. Thus, a study of how we handle F/F is worthwhile.

2.1.5. Formal path to the low-cut filter

This book defines many geophysical estimation problems. Many of them amount to statement of two goals. The first goal is a data fitting goal, the goal that the model should imply some observed data. The second goal is that the model be not too big or too wiggly. We will state these goals as two residuals, each of which is ideally zero. A very simple data fitting goal would be that the model m equals the data d, thus the difference should vanish, say $0 \approx m - d$. A more interesting goal is that the model should match the data especially at high frequencies but not necessarily at low frequencies.

$$0 \quad \approx \quad -i\omega(m - d) \tag{2.9}$$

A danger of this goal is that the model could have a zero-frequency component of infinite magnitude as well as large amplitudes for low frequencies. To suppress this,

we need the second goal, a model residual which is to be minimized. We need a small number ϵ. The model goal is

$$0 \quad \approx \quad \epsilon \, m \tag{2.10}$$

To see the consequence of these two goals, we add the squares of the residuals

$$Q(m) \quad = \quad \omega^2(m-d)^2 + \epsilon^2 m^2 \tag{2.11}$$

and then we minimize $Q(m)$ by setting its derivative to zero

$$0 \quad = \quad \frac{dQ}{dm} \quad = \quad 2\omega^2(m-d) + 2\epsilon^2 m \tag{2.12}$$

or

$$m \quad = \quad \frac{\omega^2}{\omega^2+\epsilon^2} \, d \tag{2.13}$$

which is a low-cut filter with a cutoff frequency of $\omega_0 = \epsilon$.

Of some curiosity and significance is the numerical choice of ϵ. The general theory says we need an epsilon, but does not say how much. For now let us simply rename $\epsilon = \omega_0$ and think of it as a "cut off frequency".

2.1.6. The plane-wave destructor

We address the question of shifting signals into best alignment. The most natural approach might seem to be via cross correlations. That is indeed a good approach when signals are shifted by large amounts. Here we assume signals are shifted by small amounts, often less than a single pixel. We'll take an approach closely related to differential equations. Consider this definition of a residual.

$$0 \quad \approx \quad \text{residual}(t,x) \quad = \quad \left(\frac{\partial}{\partial x} + p \frac{\partial}{\partial t} \right) u(t,x) \tag{2.14}$$

By taking derivatives we see the residual vanishes when the two-dimensional observation $u(t,x)$ matches the equation of moving waves $u(t-px)$. The parameter p has units inverse to velocity, the velocity of propagation.

In practice, $u(t,x)$ might not be a perfect wave but an observed field of many waves that we might wish to fit to the idea of a single wave of a single p. We seek the parameter p. First we need a method of discretization that allows the mesh for du/dt to overlay exactly $\partial u/\partial x$. To this end I chose to represent the t-derivative by

averaging a finite difference at x with one at $x+\Delta x$.

$$\frac{\partial u}{\partial t} \quad \approx \quad \frac{1}{2}\left(\frac{u(t+\Delta t,x)-u(t,x)}{\Delta t}\right)+\frac{1}{2}\left(\frac{u(t+\Delta t,x+\Delta x)-u(t,x+\Delta x)}{\Delta t}\right) \tag{2.15}$$

Likewise there is an analogous expression for the x-derivative with t and x interchanged. Now the difference operator $\delta_x + p\delta_t$ is a two-dimensional filter that fits on a 2×2 differencing star. We may represent equation (2.14) as a matrix operation,

$$\mathbf{0} \quad \approx \quad \mathbf{r} \quad = \quad \mathbf{A} \quad \mathbf{u} \tag{2.16}$$

where the two-dimensional convolution with the differential operator is denoted $\mathbf{A}$.

The module `wavekill()` applies the operator $a\delta_x + b\delta_t$, which can be specialized to the operators we will actually need, namely δ_x, δ_t, $\delta_x + p_i\delta_t$. wavekill

Now let us find the numerical value of p that fits a plane wave $u(t-px)$ to observations $u(t,x)$. Let $\mathbf{x}$ be an abstract vector whose components are values of $\partial u/\partial x$ taken everywhere on a 2-D mesh in (t,x). Likewise, let $\mathbf{t}$ contain $\partial u/\partial t$.

```
module wavekill_mod{
  contains
  subroutine wavekill(aa,bb,nt,nx,uu,vv){
    real   :: aa,bb(:,:),uu(:,:),vv(:,:)
    integer:: it,ix,nt,nx
    real   :: s11(nt,nx),s12(nt,nx),s21(nt,nx),s22(nt,nx)
    s11 = -aa-bb;   s12 = aa-bb
    s21 = -aa+bb;   s22 = aa+bb
    vv=0.
    do ix=1,nx-1{
      do it=1,nt-1{
        vv(it,ix)=uu(it  ,ix  )*s11(it,ix)+&
                  uu(it  ,ix+1)*s12(it,ix)+&
                  uu(it+1,ix  )*s21(it,ix)+&
                  uu(it+1,ix+1)*s22(it,ix)
        }
    }
    vv(nt,:)=vv(nt-1,:)
    vv(:,nx)=vv(:,nx-1)
  }
}
```

Back

Since we want $\mathbf{x} + p\,\mathbf{t} \approx \mathbf{0}$, we minimize the quadratic function of p,

$$Q(p) \quad = \quad (\mathbf{x} + p\,\mathbf{t}) \cdot (\mathbf{x} + p\,\mathbf{t}) \tag{2.17}$$

by setting to zero the derivative. We get

$$p \quad = \quad - \frac{\mathbf{x} \cdot \mathbf{t}}{\mathbf{t} \cdot \mathbf{t}} \tag{2.18}$$

Since data will not always fit the model very well, it may be helpful to have some way to measure how good the fit is. I suggest

$$C^2 \quad = \quad 1 - \frac{(\mathbf{x} + p\,\mathbf{t}) \cdot (\mathbf{x} + p\,\mathbf{t})}{\mathbf{x} \cdot \mathbf{x}} \tag{2.19}$$

which, on inserting $p = -(\mathbf{x} \cdot \mathbf{t})/(\mathbf{t} \cdot \mathbf{t})$, leads to C, where

$$C \quad = \quad \frac{\mathbf{x} \cdot \mathbf{t}}{\sqrt{(\mathbf{x} \cdot \mathbf{x})(\mathbf{t} \cdot \mathbf{t})}} \tag{2.20}$$

is known as the "**normalized correlation**." The program for this calculation is straightforward. The name `puck2d()` denotes *picking* on a contin*uu*m.

```
module puck2d_mod{
  use triangle_smooth
  use wavekill_mod
  contains
  subroutine puck2d(dat,coh,pp,res,boxsz,nt,nx){
    integer              :: it,ix,nt,nx
    integer, intent( in) :: boxsz
    real,    intent( in) :: dat(:,:)
    real,    intent(out) :: coh(:,:),pp(:,:),res(:,:)
    real :: dt(nt,nx),dx(nt,nx),dtdt(nt,nx),dtdx(nt,nx),dxdx(nt,nx)
    pp=0.; call wavekill(1.,pp,nt,nx,dat,dx) # space derivative
    pp=1.; call wavekill(0.,pp,nt,nx,dat,dt) #  time derivative
    dtdx = dt*dx    # (x.t)
    dxdx = dx*dx    # (x.x)
    dtdt = dt*dt    # (t.t)
    do ix=1,nx{     # smooth along time axis
      call triangle(boxsz,nt,dtdt(:,ix),dtdt(:,ix))
      call triangle(boxsz,nt,dxdx(:,ix),dxdx(:,ix))
      call triangle(boxsz,nt,dtdx(:,ix),dtdx(:,ix))
    }
    coh = sqrt( (dtdx*dtdx) / (dtdt*dxdx) )
    pp  = -dtdx / dtdt
    call wavekill(1.,pp,nt,nx,dat,res)
  }
}
```

Back

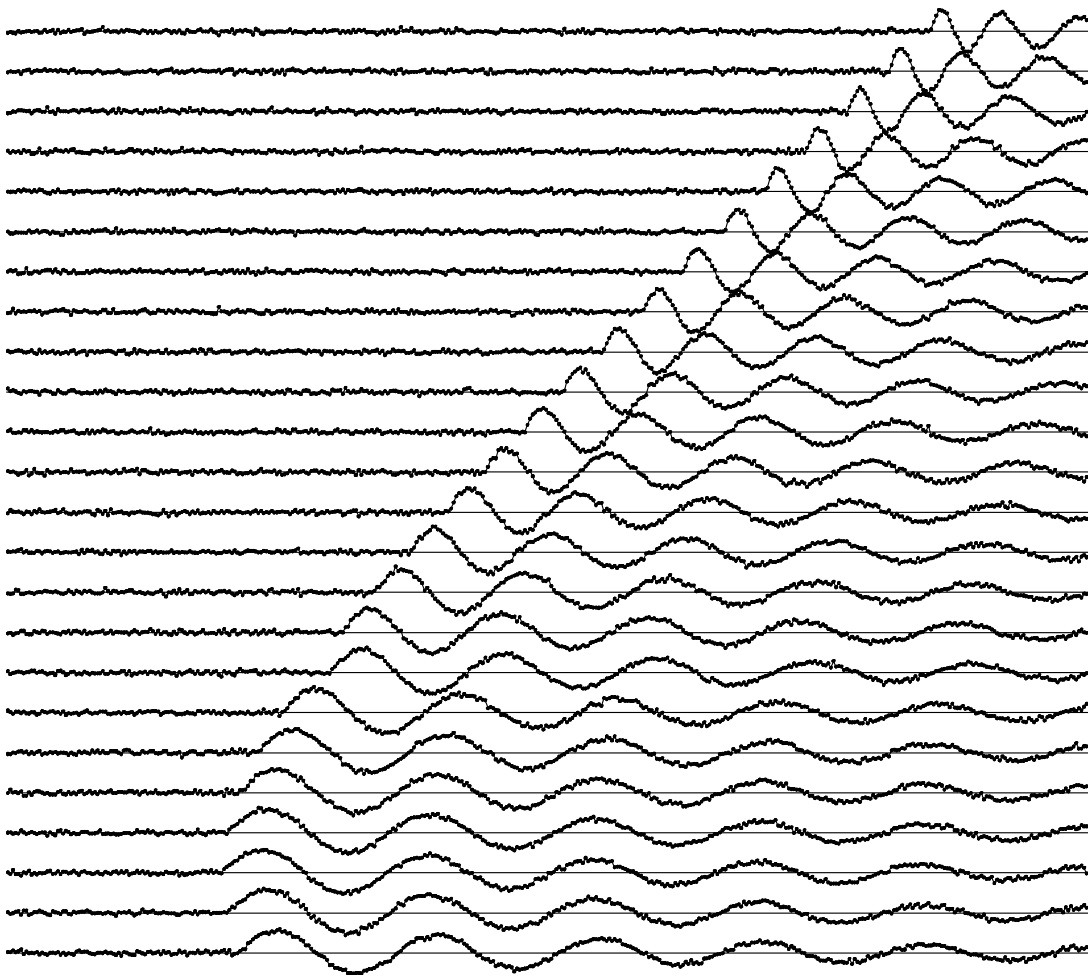


Figure 2.3: Input synthetic seismic data includes a low level of noise. lsq-puckin [ER]

Subroutine `puck2d()` below shows the code that generated Figure 2.3 through 2.5.

puck2d

An example based on synthetic data is shown in Figures 2.3-2.5. The synthetic data in Figure 2.3 mimics a reflection seismic field profile, including one trace that is slightly delayed as if recorded on a patch of unconsolidated **soil**.

Figure 2.4 shows the **residual**. The residual is small in the central region of the data; it is large where there is **spatial alias**ing; and it is large at the transient onset of the signal. The residual is rough because of the noise in the signal, because it is made from derivatives, and because the synthetic data was made by nearest-neighbor interpolation. Notice that the residual is not particularly large for the delayed trace.

Figure 2.5 shows the dips. The most significant feature of this figure is the sharp localization of the dips surrounding the delayed trace. Other methods based on "beam stacks" or Fourier concepts might lead us to conclude that the aperture must be large to resolve a wide range of angles. Here we have a narrow aperture (two traces), but the dip can change rapidly and widely.

Once the stepout $p = dt/dx$ is known between each of the signals, it is a simple matter to integrate to get the total time shift. A real-life example is shown in Figure

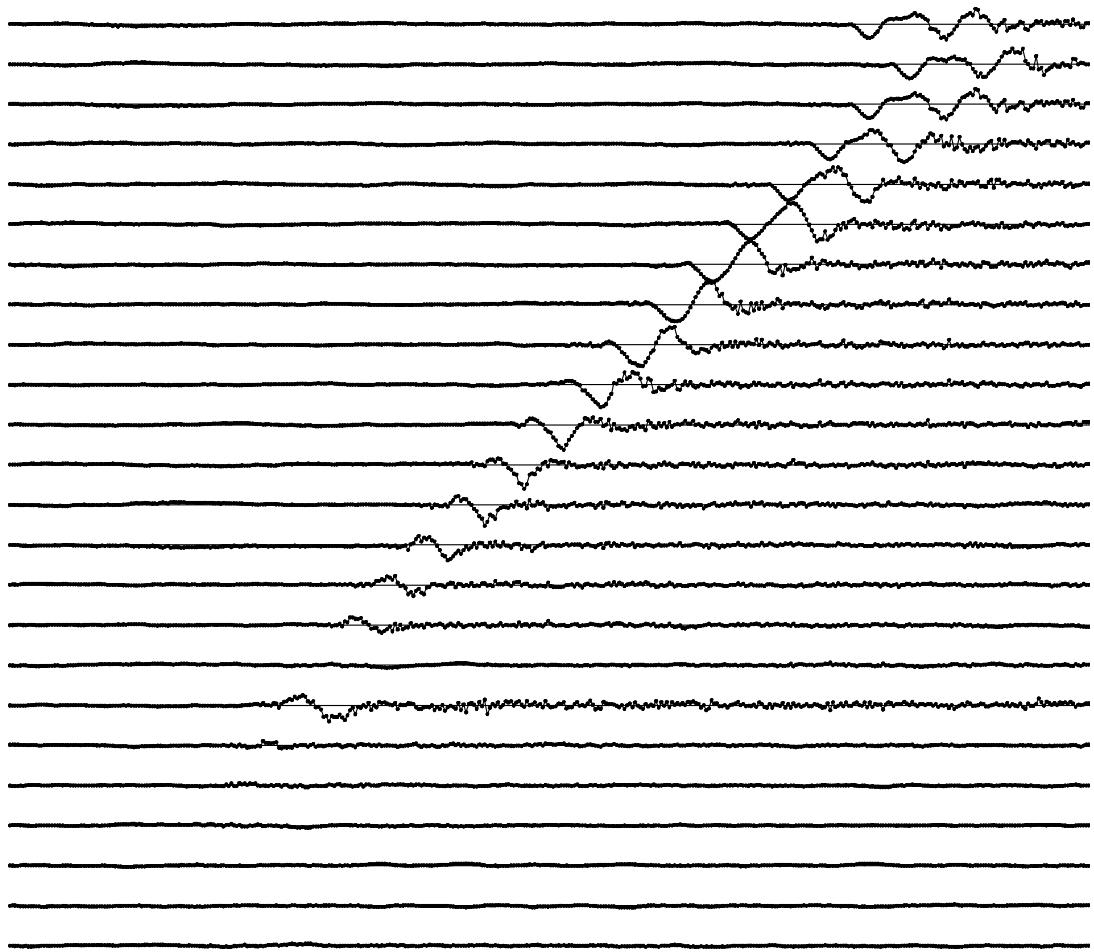


Figure 2.4: Residuals, i.e., an evaluation of $U_x + pU_t$. lsq-residual [ER]

Figure 2.5: Output values of p are shown by the slope of short line segments. lsq-puckout [ER]

2.6. In this case the flattening was a function of x only. More interesting (and more complicated) cases arise when the stepout $p = dt/dx$ is a function of both x and t. The code shown here should work well in such cases. (I should dig some up.)

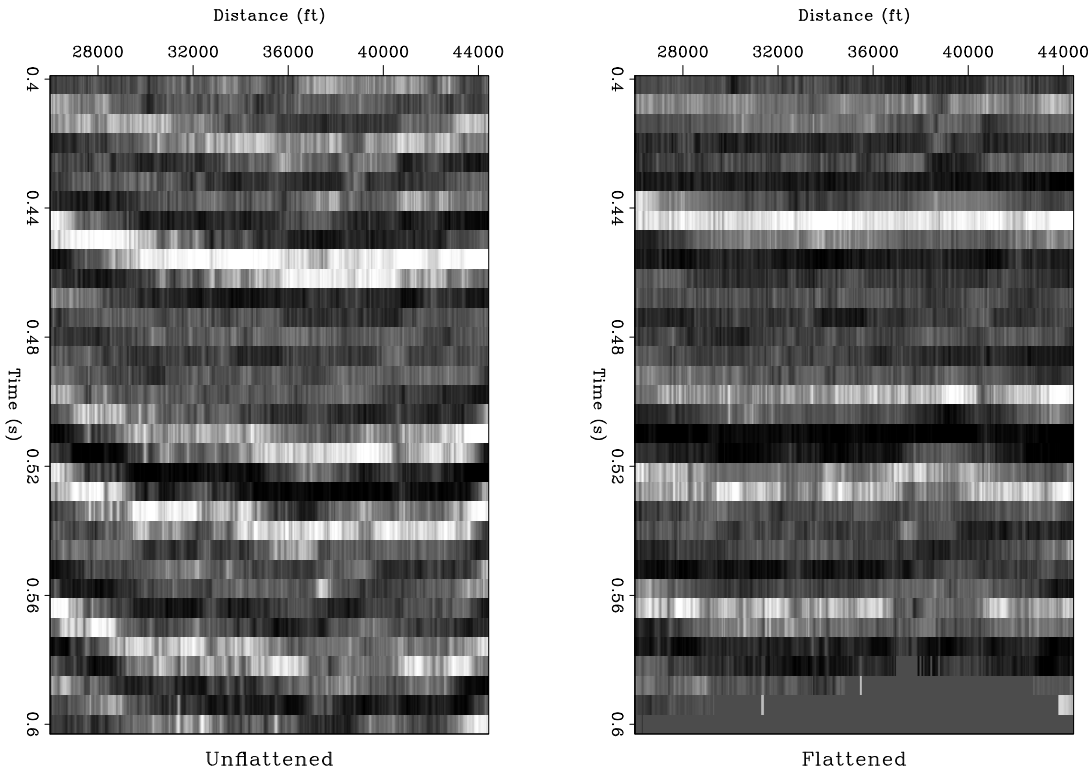


Figure 2.6: A seismic line before and after flattening. lsq-TwoD [ER,M]

A disadvantage, well known to people who routinely work with finite-difference

solutions to partial differential equations, is that for short wavelengths a finite difference operator is not the same as a differential operator; therefore the numerical value of p is biased. This problem can be overcome in the following way. First estimate the slope $p = dt/dx$ between each trace. Then shift the traces to flatten them. Now there may be a residual p because of the bias in the initial estimate of p. This process can be iterated until the data is flattened.

Everywhere in a plane we have solved a least squares problem for a single value p. In the next section we undertake to solve least squares problems for multiple parameters.

EXERCISES:

1 It is possible to reject two dips with the operator

$$(\partial_x + p_1 \partial_t)(\partial_x + p_2 \partial_t) \tag{2.21}$$

This is equivalent to

$$\left(\frac{\partial^2}{\partial x^2} + a \frac{\partial^2}{\partial x \partial t} + b \frac{\partial^2}{\partial t^2} \right) u(t,x) = v(t,x) \quad \approx \quad 0 \tag{2.22}$$

where u is the input signal and v is the output signal. Show how to solve for a and b by minimizing the energy in v.

2 Given a and b from the previous exercise, what are p_1 and p_2?

2.2. MULTIVARIATE LEAST SQUARES

2.2.1. Inside an abstract vector

In engineering uses, a vector has three scalar components that correspond to the three dimensions of the space in which we live. In least-squares data analysis, a vector is a one-dimensional array that can contain many different things. Such an array is an "**abstract vector**." For example, in earthquake studies, the vector might contain the time an earthquake began, as well as its latitude, longitude, and depth. Alternatively, the abstract vector might contain as many components as there are seismometers, and each component might be the arrival time of an earthquake wave. Used in signal analysis, the vector might contain the values of a signal at successive instants in time or, alternatively, a collection of signals. These signals might

be "**multiplex**ed" (interlaced) or "demultiplexed" (all of each signal preceding the next). When used in image analysis, the one-dimensional array might contain an image, which could itself be thought of as an array of signals. Vectors, including abstract vectors, are usually denoted by **boldface letters** such as **p** and **s**. Like physical vectors, abstract vectors are **orthogonal** when their dot product vanishes: $\mathbf{p} \cdot \mathbf{s} = 0$. Orthogonal vectors are well known in physical space; we will also encounter them in abstract vector space.

We consider first a hypothetical application with one data vector **d** and two fitting vectors $\mathbf{f}_1$ and $\mathbf{f}_2$. Each fitting vector is also known as a "**regressor**." Our first task is to approximate the data vector **d** by a scaled combination of the two regressor vectors. The scale factors x_1 and x_2 should be chosen so that the model matches the data; i.e.,

$$\mathbf{d} \quad \approx \quad \mathbf{f}_1 x_1 + \mathbf{f}_2 x_2 \tag{2.23}$$

Notice that we could take the partial derivative of the data in (2.23) with respect to an unknown, say x_1, and the result is the regressor $\mathbf{f}_1$.

The **partial derivative** of all theoretical data with respect to any model parameter gives a **regressor**. A **regressor** is a column in the matrix of partial-derivatives, $\partial d_i / \partial m_j$.

The fitting goal (2.23) is often expressed in the more compact mathematical matrix notation $\mathbf{d} \approx \mathbf{F}\mathbf{x}$, but in our derivation here we will keep track of each component explicitly and use mathematical matrix notation to summarize the final result. Fitting the observed data $\mathbf{d} = \mathbf{d}^{\text{obs}}$ to its two theoretical parts $\mathbf{f}_1 x_1$ and $\mathbf{f}_2 x_2$ can be expressed as minimizing the length of the residual vector $\mathbf{r}$, where

$$\mathbf{0} \approx \mathbf{r} = \mathbf{d}^{\text{theor}} - \mathbf{d}^{\text{obs}} \tag{2.24}$$

$$\mathbf{0} \approx \mathbf{r} = \mathbf{f}_1 x_1 + \mathbf{f}_2 x_2 - \mathbf{d} \tag{2.25}$$

We use a dot product to construct a sum of squares (also called a "**quadratic form**") of the components of the residual vector:

$$Q(x_1, x_2) = \mathbf{r} \cdot \mathbf{r} \tag{2.26}$$

$$= (\mathbf{f}_1 x_1 + \mathbf{f}_2 x_2 - \mathbf{d}) \cdot (\mathbf{f}_1 x_1 + \mathbf{f}_2 x_2 - \mathbf{d}) \tag{2.27}$$

To find the gradient of the quadratic form $Q(x_1, x_2)$, you might be tempted to expand

out the dot product into all nine terms and then differentiate. It is less cluttered, however, to remember the product rule, that

$$\frac{d}{dx}\mathbf{r}\cdot\mathbf{r} \quad = \quad \frac{d\mathbf{r}}{dx}\cdot\mathbf{r}+\mathbf{r}\cdot\frac{d\mathbf{r}}{dx} \tag{2.28}$$

Thus, the gradient of $Q(x_1,x_2)$ is defined by its two components:

$$\frac{\partial Q}{\partial x_1} \quad = \quad \mathbf{f}_1\cdot(\mathbf{f}_1x_1+\mathbf{f}_2x_2-\mathbf{d})+(\mathbf{f}_1x_1+\mathbf{f}_2x_2-\mathbf{d})\cdot\mathbf{f}_1 \tag{2.29}$$

$$\frac{\partial Q}{\partial x_2} \quad = \quad \mathbf{f}_2\cdot(\mathbf{f}_1x_1+\mathbf{f}_2x_2-\mathbf{d})+(\mathbf{f}_1x_1+\mathbf{f}_2x_2-\mathbf{d})\cdot\mathbf{f}_2 \tag{2.30}$$

Setting these derivatives to zero and using $(\mathbf{f}_1\cdot\mathbf{f}_2)=(\mathbf{f}_2\cdot\mathbf{f}_1)$ etc., we get

$$(\mathbf{f}_1\cdot\mathbf{d}) \quad = \quad (\mathbf{f}_1\cdot\mathbf{f}_1)x_1+(\mathbf{f}_1\cdot\mathbf{f}_2)x_2 \tag{2.31}$$

$$(\mathbf{f}_2\cdot\mathbf{d}) \quad = \quad (\mathbf{f}_2\cdot\mathbf{f}_1)x_1+(\mathbf{f}_2\cdot\mathbf{f}_2)x_2 \tag{2.32}$$

We can use these two equations to solve for the two unknowns x_1 and x_2. Writing

this expression in matrix notation, we have

$$\begin{bmatrix} (\mathbf{f}_1 \cdot \mathbf{d}) \\ (\mathbf{f}_2 \cdot \mathbf{d}) \end{bmatrix} = \begin{bmatrix} (\mathbf{f}_1 \cdot \mathbf{f}_1) & (\mathbf{f}_1 \cdot \mathbf{f}_2) \\ (\mathbf{f}_2 \cdot \mathbf{f}_1) & (\mathbf{f}_2 \cdot \mathbf{f}_2) \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \tag{2.33}$$

It is customary to use matrix notation without dot products. To do this, we need some additional definitions. To clarify these definitions, we inspect vectors $\mathbf{f}_1$, $\mathbf{f}_2$, and $\mathbf{d}$ of three components. Thus

$$\mathbf{F} = [\mathbf{f}_1 \quad \mathbf{f}_2] = \begin{bmatrix} f_{11} & f_{12} \\ f_{21} & f_{22} \\ f_{31} & f_{32} \end{bmatrix} \tag{2.34}$$

Likewise, the *transposed* matrix $\mathbf{F}'$ is defined by

$$\mathbf{F}' = \begin{bmatrix} f_{11} & f_{21} & f_{31} \\ f_{12} & f_{22} & f_{32} \end{bmatrix} \tag{2.35}$$

Using this matrix $\mathbf{F}'$ there is a simple expression for the gradient calculated in equation (2.29) which will be used many times throughout this book, in nearly every

example:

$$\mathbf{g} \quad = \quad \begin{bmatrix} \frac{\partial Q}{\partial x_1} \\ \frac{\partial Q}{\partial x_2} \end{bmatrix} \quad = \quad \begin{bmatrix} \mathbf{f}_1 \cdot \mathbf{r} \\ \mathbf{f}_2 \cdot \mathbf{r} \end{bmatrix} \quad = \quad \begin{bmatrix} f_{11} & f_{21} & f_{31} \\ f_{12} & f_{22} & f_{32} \end{bmatrix} \begin{bmatrix} r_1 \\ r_1 \\ r_3 \end{bmatrix} \quad = \quad \mathbf{F}'\mathbf{r} \tag{2.36}$$

In words this expression says, the gradient is found by putting the residual into the adjoint operator $\mathbf{g} = \mathbf{F}'\mathbf{r}$. Notice the gradient $\mathbf{g}$ has the same number of components as the unknown solution $\mathbf{x}$, so we can think of the gradient as a $\Delta\mathbf{x}$, something we could add to $\mathbf{x}$ getting $\mathbf{x} + \Delta\mathbf{x}$. Later we'll see how much of $\Delta\mathbf{x}$ we'll want to add to $\mathbf{x}$.

We will have reached the best solution when we find the gradient $\mathbf{g} = \mathbf{0}$ vanishes which happens, as equation (2.36) says, when the residual is orthogonal to all the fitting functions (all the rows in the matrix $\mathbf{F}'$, the columns in $\mathbf{F}$, are perpendicular to $\mathbf{r}$).

The matrix in equation (2.33) contains dot products. Matrix multiplication is

another way of representing the dot products:

$$\begin{bmatrix} (\mathbf{f}_1 \cdot \mathbf{f}_1) & (\mathbf{f}_1 \cdot \mathbf{f}_2) \\ (\mathbf{f}_2 \cdot \mathbf{f}_1) & (\mathbf{f}_2 \cdot \mathbf{f}_2) \end{bmatrix} = \begin{bmatrix} f_{11} & f_{21} & f_{31} \\ f_{12} & f_{22} & f_{32} \end{bmatrix} \begin{bmatrix} f_{11} & f_{12} \\ f_{21} & f_{22} \\ f_{31} & f_{32} \end{bmatrix} \tag{2.37}$$

Thus, equation (2.33) without dot products is

$$\begin{bmatrix} f_{11} & f_{21} & f_{31} \\ f_{12} & f_{22} & f_{32} \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ d_3 \end{bmatrix} = \begin{bmatrix} f_{11} & f_{21} & f_{31} \\ f_{12} & f_{22} & f_{32} \end{bmatrix} \begin{bmatrix} f_{11} & f_{12} \\ f_{21} & f_{22} \\ f_{31} & f_{32} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \tag{2.38}$$

which has the matrix abbreviation

$$\mathbf{F}'\mathbf{d} = (\mathbf{F}'\,\mathbf{F})\mathbf{x} \tag{2.39}$$

Equation (2.39) is the classic result of least-squares fitting of data to a collection of regressors. Obviously, the same matrix form applies when there are more than two regressors and each vector has more than three components. Equation (2.39) leads to an **analytic solution** for **x** using an inverse matrix. To solve formally for the unknown **x**, we premultiply by the inverse matrix $(\mathbf{F}'\,\mathbf{F})^{-1}$:

$$\mathbf{x} \quad = \quad (\mathbf{F}' \mathbf{F})^{-1} \mathbf{F}' \mathbf{d} \tag{2.40}$$

Equation (2.40) is the central result of **least-squares** theory. We see it everywhere.

In our first manipulation of matrix algebra, we move around some parentheses in (2.39):

$$\mathbf{F}' \mathbf{d} \quad = \quad \mathbf{F}' (\mathbf{F} \mathbf{x}) \tag{2.41}$$

Moving the parentheses implies a regrouping of terms or a reordering of a computation. You can verify the validity of moving the parentheses if you write (2.41) in full as the set of two equations it represents. Equation (2.39) led to the "analytic" solution (2.40). In a later section on conjugate directions, we will see that equation (2.41) expresses better than (2.40) the philosophy of iterative methods.

Notice how equation (2.41) invites us to cancel the matrix $\mathbf{F}'$ from each side. We cannot do that of course, because $\mathbf{F}'$ is not a number, nor is it a square matrix with an inverse. If you really want to cancel the matrix $\mathbf{F}'$, you may, but the equation

is then only an approximation that restates our original goal (2.23):

$$\mathbf{d} \quad \approx \quad \mathbf{Fx} \tag{2.42}$$

A speedy problem solver might ignore the mathematics covering the previous page, study his or her application until he or she is able to write the **statement of goals** (2.42) = (2.23), premultiply by $\mathbf{F}'$, replace $\approx$ by =, getting (2.39), and take (2.39) to a simultaneous equation-solving program to get $\mathbf{x}$.

What I call "**fitting goals**" are called "**regressions**" by statisticians. In common language the word regression means to "trend toward a more primitive perfect state" which vaguely resembles reducing the size of (energy in) the residual $\mathbf{r} = \mathbf{Fx} - \mathbf{d}$. Formally this is often written as:

$$\min_{\mathbf{x}} \; ||\mathbf{Fx} - \mathbf{d}|| \tag{2.43}$$

The notation above with two pairs of vertical lines looks like double absolute value, but we can understand it as a reminder to square and sum all the components. This formal notation is more explicit about what is constant and what is variable during the fitting.

2.2.2. Normal equations

An important concept is that when energy is minimum, the residual is orthogonal to the fitting functions. The fitting functions are the column vectors $\mathbf{f}_1$, $\mathbf{f}_2$, and $\mathbf{f}_3$. Let us verify only that the dot product $\mathbf{r} \cdot \mathbf{f}_2$ vanishes; to do this, we'll show that those two vectors are orthogonal. Energy minimum is found by

$$0 \quad = \quad \frac{\partial}{\partial x_2} \mathbf{r} \cdot \mathbf{r} \quad = \quad 2\,\mathbf{r} \cdot \frac{\partial \mathbf{r}}{\partial x_2} \quad = \quad 2\,\mathbf{r} \cdot \mathbf{f}_2 \tag{2.44}$$

(To compute the derivative refer to equation (2.25).) Equation (2.44) shows that the residual is orthogonal to a fitting function. The fitting functions are the column vectors in the fitting matrix.

The basic least-squares equations are often called the "**normal**" equations. The word "normal" means perpendicular. We can rewrite equation (2.41) to emphasize the perpendicularity. Bring both terms to the left, and recall the definition of the residual $\mathbf{r}$ from equation (2.25):

$$\mathbf{F}'(\mathbf{F}\mathbf{x} - \mathbf{d}) \quad = \quad \mathbf{0} \tag{2.45}$$

$$\mathbf{F}'\mathbf{r} \quad = \quad \mathbf{0} \tag{2.46}$$

Equation (2.46) says that the **residual** vector $\mathbf{r}$ is perpendicular to each row in the $\mathbf{F}'$ matrix. These rows are the **fitting function**s. Therefore, the residual, after it has been minimized, is perpendicular to *all* the fitting functions.

2.2.3. Differentiation by a complex vector

Complex numbers frequently arise in physical problems, particularly those with Fourier series. Let us extend the multivariable least-squares theory to the use of complex-valued unknowns $\mathbf{x}$. First recall how complex numbers were handled with single-variable least squares; i.e., as in the discussion leading up to equation (2.5). Use a prime, such as $\mathbf{x}'$, to denote the complex conjugate of the transposed vector $\mathbf{x}$. Now write the positive **quadratic form** as

$$Q(\mathbf{x}',\mathbf{x}) \quad = \quad (\mathbf{F}\mathbf{x}-\mathbf{d})'(\mathbf{F}\mathbf{x}-\mathbf{d}) \quad = \quad (\mathbf{x}'\mathbf{F}'-\mathbf{d}')(\mathbf{F}\mathbf{x}-\mathbf{d}) \tag{2.47}$$

After equation (2.4), we minimized a quadratic form $Q(\bar{X},X)$ by setting to zero both $\partial Q/\partial X$ and $\partial Q/\partial \bar{X}$. We noted that only one of $\partial Q/\partial X$ and $\partial Q/\partial \bar{X}$ is necessarily zero because they are conjugates of each other. Now take the derivative of Q with respect to the (possibly complex, row) vector $\mathbf{x}'$. Notice that $\partial Q/\partial \mathbf{x}'$ is the complex

conjugate transpose of $\partial Q/\partial \mathbf{x}$. Thus, setting one to zero sets the other also to zero. Setting $\partial Q/\partial \mathbf{x}' = \mathbf{0}$ gives the normal equations:

$$\mathbf{0} \quad = \quad \frac{\partial Q}{\partial \mathbf{x}'} \quad = \quad \mathbf{F}'(\mathbf{F}\mathbf{x} - \mathbf{d}) \tag{2.48}$$

The result is merely the complex form of our earlier result (2.45). Therefore, differentiating by a complex vector is an abstract concept, but it gives the same set of equations as differentiating by each scalar component, and it saves much clutter.

2.2.4. From the frequency domain to the time domain

Equation (2.4) is a frequency-domain quadratic form that we minimized by varying a single parameter, a Fourier coefficient. Now we will look at the same problem in the time domain. We will see that the time domain offers flexibility with boundary conditions, constraints, and weighting functions. The notation will be that a filter f_t has input x_t and output y_t. In Fourier space this is $Y = XF$. There are two problems to look at, unknown filter F and unknown input X.

- **Unknown filter**

When inputs and outputs are given, the problem of finding an unknown filter appears to be overdetermined, so we write $\mathbf{y} \approx \mathbf{Xf}$ where the matrix $\mathbf{X}$ is a matrix of downshifted columns like (1.5). Thus the quadratic form to be minimized is a restatement of equation (2.47) with filter definitions:

$$Q(\mathbf{f}', \mathbf{f}) \quad = \quad (\mathbf{Xf} - \mathbf{y})'(\mathbf{Xf} - \mathbf{y}) \tag{2.49}$$

The solution $\mathbf{f}$ is found just as we found (2.48), and it is the set of simultaneous equations $\mathbf{0} = \mathbf{X}'(\mathbf{Xf} - \mathbf{y})$.

- **Unknown input: deconvolution with a known filter**

For solving the unknown-input problem, we put the known filter f_t in a matrix of downshifted columns $\mathbf{F}$. Our statement of wishes is now to find x_t so that $\mathbf{y} \approx \mathbf{Fx}$. We can expect to have trouble finding unknown inputs x_t when we are dealing with certain kinds of filters, such as **bandpass filter**s. If the output is zero in a frequency band, we will never be able to find the input in that band and will need to prevent x_t from diverging there. We do this by the statement that we wish $\mathbf{0} \approx \epsilon\,\mathbf{x}$, where ϵ is a parameter that is small and whose exact size will be chosen by experimentation.

Putting both wishes into a single, partitioned matrix equation gives

$$\begin{bmatrix} \mathbf{0} \\ \mathbf{0} \end{bmatrix} \quad \approx \quad \begin{bmatrix} \mathbf{r}_1 \\ \mathbf{r}_2 \end{bmatrix} \quad = \quad \begin{bmatrix} \mathbf{F} \\ \epsilon\, \mathbf{I} \end{bmatrix} \mathbf{x} \quad - \quad \begin{bmatrix} \mathbf{y} \\ \mathbf{0} \end{bmatrix} \tag{2.50}$$

To minimize the residuals $\mathbf{r}_1$ and $\mathbf{r}_2$, we can minimize the scalar $\mathbf{r}'\mathbf{r} = \mathbf{r}'_1\mathbf{r}_1 + \mathbf{r}'_2\mathbf{r}_2$. This is

$$\begin{aligned} Q(\mathbf{x}', \mathbf{x}) &= (\mathbf{F}\mathbf{x} - \mathbf{y})'(\mathbf{F}\mathbf{x} - \mathbf{y}) + \epsilon^2 \mathbf{x}'\mathbf{x} \\ &= (\mathbf{x}'\mathbf{F}' - \mathbf{y}')(\mathbf{F}\mathbf{x} - \mathbf{y}) + \epsilon^2 \mathbf{x}'\mathbf{x} \end{aligned} \tag{2.51}$$

We solved this minimization in the frequency domain (beginning from equation (2.4)).

Formally the solution is found just as with equation (2.48), but this solution looks unappealing in practice because there are so many unknowns and because the problem can be solved much more quickly in the Fourier domain. To motivate ourselves to solve this problem in the time domain, we need either to find an approximate solution method that is much faster, or to discover that constraints or time-variable weighting functions are required in some applications. This is an issue we must be continuously alert to, whether the cost of a method is justified by its need.

EXERCISES:

1 In 1695, 150 years before Lord Kelvin's absolute temperature scale, 120 years before Sadi Carnot's PhD thesis, 40 years before Anders Celsius, and 20 years before Gabriel Farenheit, the French physicist Guillaume **Amontons**, deaf since birth, took a mercury manometer (pressure gauge) and sealed it inside a glass pipe (a constant volume of air). He heated it to the boiling point of water at 100°C. As he lowered the temperature to freezing at 0° C, he observed the pressure dropped by 25% . He could not drop the temperature any further but he supposed that if he could drop it further by a factor of three, the pressure would drop to zero (the lowest possible pressure) and the temperature would have been the lowest possible temperature. Had he lived after Anders Celsius he might have calculated this temperature to be $-300°$ C (Celsius). Absolute zero is now known to be $-273°$ C.

It is your job to be Amontons' lab assistant. Your ith measurement of temperature T_i you make with Issac Newton's thermometer and you measure pressure P_i and volume V_i in the metric system. Amontons needs you to fit his data with the regression $0 \approx \alpha(T_i - T_0) - P_i V_i$ and calculate the temperature shift T_0 that Newton should have made when he defined his temperature scale. Do not solve

this problem! Instead, cast it in the form of equation (2.23), identifying the data **d** and the two column vectors $\mathbf{f}_1$ and $\mathbf{f}_2$ that are the fitting functions. Relate the model parameters x_1 and x_2 to the physical parameters α and T_0. Suppose you make ALL your measurements at room temperature, can you find T_0? Why or why not?

2.3. KRYLOV SUBSPACE ITERATIVE METHODS

The **solution time** for simultaneous **linear equations** grows cubically with the number of unknowns. There are three regimes for solution; which one is applicable depends on the number of unknowns m. For m three or less, we use analytical methods. We also sometimes use analytical methods on matrices of size 4×4 when the matrix contains enough zeros. Today in year 2001, a deskside workstation, working an hour solves about a 4000×4000 set of simultaneous equations. A square image packed into a 4096 point vector is a 64×64 array. The computer power for linear algebra to give us solutions that fit in a $k \times k$ image is thus proportional to k^6, which

means that even though computer power grows rapidly, imaging resolution using "exact numerical methods" hardly grows at all from our 64×64 current practical limit.

The retina in our eyes captures an image of size about 1000×1000 which is a lot bigger than 64×64. Life offers us many occasions where final images exceed the 4000 points of a 64×64 array. To make linear algebra (and inverse theory) relevant to such problems, we investigate special techniques. A numerical technique known as the "**conjugate-direction method**" works well for all values of m and is our subject here. As with most simultaneous equation solvers, an exact answer (assuming exact arithmetic) is attained in a finite number of steps. And if n and m are too large to allow enough iterations, the iterative methods can be interrupted at any stage, the partial result often proving useful. Whether or not a partial result actually is useful is the subject of much research; naturally, the results vary from one application to the next.

2.3.1. Sign convention

On the last day of the survey, a storm blew up, the sea got rough, and the receivers drifted further downwind. The data recorded that day had a larger than usual difference from that predicted by the final model. We could call $(\mathbf{d}-\mathbf{Fm})$ the ***experimental error***. (Here **d** is data, **m** is model parameters, and **F** is their linear relation).

The alternate view is that our theory was too simple. It lacked model parameters for the waves and the drifting cables. Because of this model oversimplification we had a ***modeling error*** of the opposite polarity $(\mathbf{Fm}-\mathbf{d})$.

A strong experimentalist prefers to think of the error as experimental error, something for him or her to work out. Likewise a strong analyst likes to think of the error as a theoretical problem. (Weaker investigators might be inclined to take the opposite view.)

Regardless of the above, and opposite to common practice, I define the **sign convention** for the error (or residual) as $(\mathbf{Fm}-\mathbf{d})$. When we choose this sign convention, our hazard for analysis errors will be reduced because **F** is often complicated and formed by combining many parts.

Beginners often feel disappointment when the data does not fit the model very well. They see it as a defect in the data instead of an opportunity to design a stronger theory.

2.3.2. Method of random directions and steepest descent

Let us minimize the sum of the squares of the components of the **residual** vector given by

$$\text{residual} = \text{transform} \quad \text{model space} - \text{data space} \tag{2.52}$$

$$\begin{bmatrix} \\ \mathbf{r} \\ \\ \end{bmatrix} = \begin{bmatrix} \\ \mathbf{F} \\ \\ \end{bmatrix} \begin{bmatrix} \\ \mathbf{x} \\ \\ \end{bmatrix} - \begin{bmatrix} \\ \mathbf{d} \\ \\ \end{bmatrix} \tag{2.53}$$

A **contour plot** is based on an altitude function of space. The altitude is the

dot product $\mathbf{r} \cdot \mathbf{r}$. By finding the lowest altitude, we are driving the residual vector $\mathbf{r}$ as close as we can to zero. If the residual vector $\mathbf{r}$ reaches zero, then we have solved the simultaneous equations $\mathbf{d} = \mathbf{F}\mathbf{x}$. In a two-dimensional world the vector $\mathbf{x}$ has two components, (x_1, x_2). A contour is a curve of constant $\mathbf{r} \cdot \mathbf{r}$ in (x_1, x_2)-space. These contours have a statistical interpretation as contours of uncertainty in (x_1, x_2), with measurement errors in $\mathbf{d}$.

Let us see how a random search-direction can be used to reduce the residual $0 \approx \mathbf{r} = \mathbf{F}\mathbf{x} - \mathbf{d}$. Let $\Delta\mathbf{x}$ be an abstract vector with the same number of components as the solution $\mathbf{x}$, and let $\Delta\mathbf{x}$ contain arbitrary or random numbers. We add an unknown quantity α of vector $\Delta\mathbf{x}$ to the vector $\mathbf{x}$, and thereby create $\mathbf{x}_{\text{new}}$:

$$\mathbf{x}_{\text{new}} = \mathbf{x} + \alpha \Delta\mathbf{x} \tag{2.54}$$

This gives a new residual:

$$\mathbf{r}_{\text{new}} = \mathbf{F}\ \mathbf{x}_{\text{new}} - \mathbf{d} \tag{2.55}$$

$$\mathbf{r}_{\text{new}} = \mathbf{F}(\mathbf{x} + \alpha \Delta\mathbf{x}) - \mathbf{d} \tag{2.56}$$

$$\mathbf{r}_{\text{new}} = \mathbf{r} + \alpha \Delta\mathbf{r} = (\mathbf{F}\mathbf{x} - \mathbf{d}) + \alpha \mathbf{F} \Delta\mathbf{x} \tag{2.57}$$

which defines $\Delta\mathbf{r} = \mathbf{F}\Delta\mathbf{x}$.

Next we adjust α to minimize the dot product: $\mathbf{r}_{\text{new}} \cdot \mathbf{r}_{\text{new}}$

$$(\mathbf{r}+\alpha\Delta\mathbf{r})\cdot(\mathbf{r}+\alpha\Delta\mathbf{r}) \quad = \quad \mathbf{r}\cdot\mathbf{r}+2\alpha(\mathbf{r}\cdot\Delta\mathbf{r}) \; + \; \alpha^2\Delta\mathbf{r}\cdot\Delta\mathbf{r} \tag{2.58}$$

Set to zero its derivative with respect to α using the chain rule

$$0 \quad = \quad (\mathbf{r}+\alpha\Delta\mathbf{r})\cdot\Delta\mathbf{r} \; + \; \Delta\mathbf{r}\cdot(\mathbf{r}+\alpha\Delta\mathbf{r}) \quad = \quad 2(\mathbf{r}+\alpha\Delta\mathbf{r})\cdot\Delta\mathbf{r} \tag{2.59}$$

which says that the new residual $\mathbf{r}_{\text{new}} = \mathbf{r}+\alpha\Delta\mathbf{r}$ is perpendicular to the "fitting function" $\Delta\mathbf{r}$. Solving gives the required value of α.

$$\alpha \quad = \quad -\frac{(\mathbf{r}\cdot\Delta\mathbf{r})}{(\Delta\mathbf{r}\cdot\Delta\mathbf{r})} \tag{2.60}$$

A "computation **template**" for the method of random directions is

$\mathbf{r} \longleftarrow \mathbf{Fx}-\mathbf{d}$
iterate {
$\Delta\mathbf{x} \longleftarrow$ random numbers
$\Delta\mathbf{r} \longleftarrow \mathbf{F}\,\Delta\mathbf{x}$
$\alpha \longleftarrow -(\mathbf{r}\cdot\Delta\mathbf{r})/(\Delta\mathbf{r}\cdot\Delta\mathbf{r})$
$\mathbf{x} \longleftarrow \mathbf{x}+\alpha\Delta\mathbf{x}$
$\mathbf{r} \longleftarrow \mathbf{r}+\alpha\Delta\mathbf{r}$

}

A nice thing about the method of random directions is that you do not need to know the adjoint operator $\mathbf{F}'$.

In practice, random directions are rarely used. It is more common to use the **gradient** direction than a random direction. Notice that a vector of the size of $\Delta\mathbf{x}$ is

$$\mathbf{g} \quad = \quad \mathbf{F}'\mathbf{r} \tag{2.61}$$

Notice also that this vector can be found by taking the gradient of the size of the residuals:

$$\frac{\partial}{\partial \mathbf{x}'}\, \mathbf{r}\cdot\mathbf{r} \quad = \quad \frac{\partial}{\partial \mathbf{x}'}\, (\mathbf{x}'\mathbf{F}' \,-\, \mathbf{d}')(\mathbf{F}\mathbf{x} \,-\, \mathbf{d}) \quad = \quad \mathbf{F}'\,\mathbf{r} \tag{2.62}$$

Choosing $\Delta\mathbf{x}$ to be the gradient vector $\Delta\mathbf{x} = \mathbf{g} = \mathbf{F}'\mathbf{r}$ is called "the method of **steepest descent**."

Starting from a model $\mathbf{x} = \mathbf{m}$ (which may be zero), below is a **template** of pseudocode for minimizing the residual $\mathbf{0} \approx \mathbf{r} = \mathbf{F}\mathbf{x} - \mathbf{d}$ by the steepest-descent method:

$\mathbf{r} \quad \longleftarrow \quad \mathbf{F}\mathbf{x} - \mathbf{d}$
iterate {

$$
\begin{aligned}
\Delta \mathbf{x} &\longleftarrow \mathbf{F}' \, \mathbf{r} \\
\Delta \mathbf{r} &\longleftarrow \mathbf{F} \, \Delta \mathbf{x} \\
\alpha &\longleftarrow -(\mathbf{r} \cdot \Delta \mathbf{r}) / (\Delta \mathbf{r} \cdot \Delta \mathbf{r}) \\
\mathbf{x} &\longleftarrow \mathbf{x} + \alpha \, \Delta \mathbf{x} \\
\mathbf{r} &\longleftarrow \mathbf{r} + \alpha \, \Delta \mathbf{r} \\
&\}
\end{aligned}
$$

2.3.3. Null space and iterative methods

In applications where we fit $\mathbf{d} \approx \mathbf{F}\mathbf{x}$, there might exist a vector (or a family of vectors) defined by the condition $\mathbf{0} = \mathbf{F}\mathbf{x}_{\text{null}}$. This family is called a **null space**. For example, if the operator $\mathbf{F}$ is a time derivative, then the null space is the constant function; if the operator is a second derivative, then the null space has two components, a constant function and a linear function, or combinations of them. The null space is a family of model components that have no effect on the data.

When we use the steepest-descent method, we iteratively find solutions by this

updating:

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \alpha \Delta \mathbf{x} \tag{2.63}$$

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \alpha \mathbf{F}' \mathbf{r} \tag{2.64}$$

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \alpha \mathbf{F}' (\mathbf{F}\mathbf{x} - \mathbf{d}) \tag{2.65}$$

After we have iterated to convergence, the gradient $\Delta \mathbf{x}$ vanishes as does $\mathbf{F}'(\mathbf{F}\mathbf{x} - \mathbf{d})$. Thus, an iterative solver gets the same solution as the long-winded theory leading to equation (2.40).

Suppose that by adding a huge amount of $\mathbf{x}_{\text{null}}$, we now change $\mathbf{x}$ and continue iterating. Notice that $\Delta \mathbf{x}$ remains zero because $\mathbf{F}\mathbf{x}_{\text{null}}$ vanishes. Thus we conclude that any null space in the initial guess $\mathbf{x}_0$ will remain there unaffected by the gradient-descent process.

Linear algebra theory enables us to dig up the entire null space should we so desire. On the other hand, the computer demands might be vast. Even the memory for holding the many $\mathbf{x}$ vectors could be prohibitive. A much simpler and more practical goal is to find out if the null space has any members, and if so, to view some of them. To try to see a member of the null space, we take two starting guesses and we run our iterative solver for each of them. If the two solutions, $\mathbf{x}_1$

and $\mathbf{x}_2$, are the same, there is no null space. If the solutions differ, the difference is a member of the null space. Let us see why: Suppose after iterating to minimum residual we find

$$\mathbf{r}_1 = \mathbf{F}\mathbf{x}_1 - \mathbf{d} \tag{2.66}$$

$$\mathbf{r}_2 = \mathbf{F}\mathbf{x}_2 - \mathbf{d} \tag{2.67}$$

We know that the residual squared is a convex quadratic function of the unknown $\mathbf{x}$. Mathematically that means the minimum value is unique, so $\mathbf{r}_1 = \mathbf{r}_2$. Subtracting we find $0 = \mathbf{r}_1 - \mathbf{r}_2 = \mathbf{F}(\mathbf{x}_1 - \mathbf{x}_2)$ proving that $\mathbf{x}_1 - \mathbf{x}_2$ is a model in the null space. Adding $\mathbf{x}_1 - \mathbf{x}_2$ to any to any model $\mathbf{x}$ will not change the theoretical data. Are you having trouble visualizing $\mathbf{r}$ being unique, but $\mathbf{x}$ not being unique? Imagine that $\mathbf{r}$ happens to be independent of one of the components of $\mathbf{x}$. That component is nonunique. More generally, it is some linear combination of components of $\mathbf{x}$ that $\mathbf{r}$ is independent of.

A practical way to learn about the existence of null spaces and their general appearance is simply to try gradient-descent methods beginning from various different starting guesses.

"Did I fail to run my iterative solver long enough?" is a question you might have. If two residuals from two starting solutions are not equal, $\mathbf{r}_1 \neq \mathbf{r}_2$, then you should be running your solver through more iterations.

> If two different starting solutions produce two different residuals, then you didn't run your solver through enough iterations.

2.3.4. Why steepest descent is so slow

Before we can understand why the **conjugate-direction method** is so fast, we need to see why the **steepest-descent method** is so slow. Imagine yourself sitting on the edge of a circular bowl. If you jump off the rim, you'll slide straight to the bottom at the center. Now imagine an ellipsoidal bowl of very large ellipticity. As you jump off the rim, you'll first move in the direction of the gradient. This is not towards the bottom at the center of the ellipse (unless you were sitting on the major or minor axis).

We can formalize the situation. A parametric equation for a line is $\mathbf{x} = \mathbf{x}_{\text{old}} + \alpha \Delta \mathbf{x}$ where α is the parameter for moving on the line. The process of selecting

α is called "**line search**." Think of a two-dimensional example where the vector of unknowns $\mathbf{x}$ has just two components, x_1 and x_2. Then the size of the residual vector $\mathbf{r} \cdot \mathbf{r}$ can be displayed with a contour plot in the plane of (x_1, x_2). Our ellipsoidal bowl has ellipsoidal contours of constant altitude. As we move in a line across this space by adjusting α, equation(2.58) gives our altitude. This equation has a unique minimum because it is a parabola in α. As we approach the minimum, our trajectory becomes tangential to a contour line in (x_1, x_2)-space. This is where we stop. Now we compute our new residual $\mathbf{r}$ and we compute the new gradient $\Delta\mathbf{x} = \mathbf{g} = \mathbf{F}'\mathbf{r}$. OK, we are ready for the next slide down. When we turn ourselves from "parallel to a contour line" to the direction of $\Delta\mathbf{x}$ which is "perpendicular to that contour", we are turning 90°. Our path to the bottom of the bowl will be made of many segments, each turning 90° from the previous. We will need an infinite number of such steps to reach the bottom. It happens that the amazing conjugate-direction method would reach the bottom in just two jumps (because (x_1, x_2) is a two dimensional space.)

2.3.5. Conjugate direction

In the **conjugate-direction method**, not a line, but rather a plane, is searched. A plane is made from an arbitrary linear combination of two vectors. One vector will be chosen to be the gradient vector, say $\mathbf{g}$. The other vector will be chosen to be the previous descent step vector, say $\mathbf{s} = \mathbf{x}_j - \mathbf{x}_{j-1}$. Instead of $\alpha\,\mathbf{g}$ we need a linear combination, say $\alpha\mathbf{g} + \beta\mathbf{s}$. For minimizing quadratic functions the plane search requires only the solution of a two-by-two set of linear equations for α and β. The equations will be specified here along with the program. (For *nonquadratic* functions a plane search is considered intractable, whereas a line search proceeds by bisection.)

For use in linear problems, the conjugate-direction method described in this book follows an identical path with the more well-known conjugate-gradient method. We use the conjugate-direction method for convenience in exposition and programming.

The simple form of the conjugate-direction algorithm covered here is a sequence of steps. In each step the minimum is found in the plane given by two vectors: the gradient vector and the vector of the previous step. Given the linear operator $\mathbf{F}$ and a generator of solution steps (random in the case of random directions

or gradient in the case of steepest descent), we can construct an optimally convergent iteration process, which finds the solution in no more than n steps, where n is the size of the problem. This result should not be surprising. If **F** is represented by a full matrix, then the cost of direct inversion is proportional to n^3, and the cost of matrix multiplication is n^2. Each step of an iterative method boils down to a matrix multiplication. Therefore, we need at least n steps to arrive at the exact solution. Two circumstances make large-scale optimization practical. First, for sparse convolution matrices the cost of matrix multiplication is n instead of n^2. Second, we can often find a reasonably good solution after a limited number of iterations. If both these conditions are met, the cost of optimization grows linearly with n, which is a practical rate even for very large problems. Fourier-transformed variables are often capitalized. This convention will be helpful here, so in this subsection only, we capitalize vectors transformed by the **F** matrix. As everywhere, a matrix such as **F** is printed in **boldface** type but in this subsection, vectors are *not* printed in boldface print. Thus we define the solution, the solution step (from one iteration to the next),

and the gradient by

$$X = \mathbf{F}\, x \qquad \text{solution} \tag{2.68}$$

$$S_j = \mathbf{F}\, s_j \qquad \text{solution step} \tag{2.69}$$

$$G_j = \mathbf{F}\, g_j \qquad \text{solution gradient} \tag{2.70}$$

A linear combination in solution space, say $s+g$, corresponds to $S+G$ in the conjugate space, because $S+G = \mathbf{F}s + \mathbf{F}g = \mathbf{F}(s+g)$. According to equation (2.53), the residual is the theoretical data minus the observed data.

$$R = \mathbf{F}x - D = X - D \tag{2.71}$$

The solution x is obtained by a succession of steps s_j, say

$$x = s_1 + s_2 + s_3 + \cdots \tag{2.72}$$

The last stage of each iteration is to update the solution and the residual:

$$\text{solution update :} \qquad x \leftarrow x + s \tag{2.73}$$

$$\text{residual update :} \qquad R \leftarrow R + S \tag{2.74}$$

The *gradient* vector g is a vector with the same number of components as the solution vector x. A vector with this number of components is

$$g = \mathbf{F}' R = \text{gradient} \tag{2.75}$$

$$G = \mathbf{F}\, g = \text{conjugate gradient} \tag{2.76}$$

The gradient g in the transformed space is G, also known as the **conjugate gradient**.

The minimization (2.58) is now generalized to scan not only the line with α, but simultaneously another line with β. The combination of the two lines is a plane:

$$Q(\alpha,\beta) = (R+\alpha G+\beta S)\cdot(R+\alpha G+\beta S) \tag{2.77}$$

The minimum is found at $\partial Q/\partial\alpha = 0$ and $\partial Q/\partial\beta = 0$, namely,

$$0 = G\cdot(R+\alpha G+\beta S) \tag{2.78}$$

$$0 = S\cdot(R+\alpha G+\beta S) \tag{2.79}$$

The solution is

$$\begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \frac{-1}{(G\cdot G)(S\cdot S)-(G\cdot S)^2} \begin{bmatrix} (S\cdot S) & -(S\cdot G) \\ -(G\cdot S) & (G\cdot G) \end{bmatrix} \begin{bmatrix} (G\cdot R) \\ (S\cdot R) \end{bmatrix} \tag{2.80}$$

This may look complicated. The steepest descent method requires us to compute only the two dot products $\mathbf{r} \cdot \Delta\mathbf{r}$ and $\Delta\mathbf{r} \cdot \Delta\mathbf{r}$ while equation (2.77) contains five dot products, but the extra trouble is well worth while because the "conjugate direction" is such a much better direction than the gradient direction.

The many applications in this book all need to find α and β with (2.80) and then update the solution with (2.73) and update the residual with (2.74). Thus we package these activities in a subroutine named `cgstep()`. To use that subroutine we will have a computation **template** like we had for steepest descents, except that we will have the repetitive work done by subroutine `cgstep()`. This template (or pseudocode) for minimizing the residual $\mathbf{0} \approx \mathbf{r} = \mathbf{F}\mathbf{x} - \mathbf{d}$ by the conjugate-direction method is

$$
\begin{aligned}
&\mathbf{r} \quad \longleftarrow \quad \mathbf{F}\mathbf{x} - \mathbf{d} \\
&\text{iterate } \{ \\
&\qquad \Delta\mathbf{x} \quad \longleftarrow \quad \mathbf{F}' \, \mathbf{r} \\
&\qquad \Delta\mathbf{r} \quad \longleftarrow \quad \mathbf{F} \, \Delta\mathbf{x} \\
&\qquad (\mathbf{x}, \mathbf{r}) \quad \longleftarrow \quad \text{cgstep}(\mathbf{x}, \Delta\mathbf{x}, \mathbf{r}, \Delta\mathbf{r}) \\
&\qquad \}
\end{aligned}
$$

where the subroutine `cgstep()` remembers the previous iteration and works out the

step size and adds in the proper proportion of the $\Delta\mathbf{x}$ of the previous step.

2.3.6. Routine for one step of conjugate-direction descent

Because **Fortran** does not recognize the difference between upper- and lower-case letters, the conjugate vectors G and S in the program are denoted by `gg` and `ss`. The inner part of the conjugate-direction task is in function `cgstep()`. cgstep

Observe the `cgstep()` function has a logical parameter called `forget`. This parameter does not need to be input. In the normal course of things, `forget` will be true on the first iteration and false on subsequent iterations. This refers to the fact that on the first iteration, there is no previous step, so the conjugate direction method is reduced to the steepest descent method. At any iteration, however, you have the option to set `forget=.true.` which amounts to restarting the calculation from the current location, something we rarely find reason to do.

```
module cgstep_mod  {
    real, dimension (:), allocatable, private   :: s, ss
contains
    integer function cgstep( forget, x, g, rr, gg)  {
        real, dimension (:)  :: x, g, rr, gg
        logical              :: forget
        double precision     :: sds, gdg, gds, determ, gdr, sdr, alfa, beta
        if( .not. allocated (s)) {   forget = .true.
                    allocate ( s (size ( x)))
                    allocate (ss (size (rr)))
                    }
        if( forget){ s = 0.;  ss = 0.;  beta = 0.d0    # steepest descent
                    if( dot_product(gg, gg) == 0 )
                              call erexit('cgstep: grad vanishes identically')
                    alfa = - sum( dprod( gg, rr)) / sum( dprod( gg, gg))
                    }
        else{ gdg = sum( dprod( gg, gg))       # search plane by solving 2-by-2
              sds = sum( dprod( ss, ss))       #  G . (R - G*alfa - S*beta) = 0
              gds = sum( dprod( gg, ss))       #  S . (R - G*alfa - S*beta) = 0
              if( gdg==0. .or. sds==0.)  { cgstep = 1;  return }
              determ = gdg * sds * max (1.d0 - (gds/gdg)*(gds/sds), 1.d-12)
              gdr = - sum( dprod( gg, rr))
              sdr = - sum( dprod( ss, rr))
              alfa = ( sds * gdr - gds * sdr ) / determ
              beta = (-gds * gdr + gdg * sdr ) / determ
              }
        s  = alfa *  g + beta *  s            # update solution step
        ss = alfa * gg + beta * ss            # update residual step
        x  =  x +  s                          # update solution
        rr = rr + ss                          # update residual
        forget = .false.;   cgstep = 0
    }
    subroutine cgstep_close ( ) {
         if( allocated( s))  deallocate( s, ss)
    }
}
```

Back

2.3.7. A basic solver program

There are many different methods for iterative least-square estimation some of which will be discussed later in this book. The conjugate-gradient (CG) family (including the first order conjugate-direction method described above) share the property that theoretically they achieve the solution in n iterations, where n is the number of unknowns. The various CG methods differ in their numerical errors, memory required, adaptability to non-linear optimization, and their requirements on accuracy of the adjoint. What we do in this section is to show you the generic interface.

None of us is an expert in both geophysics and in optimization theory (OT), yet we need to handle both. We would like to have each group write its own code with a relatively easy interface. The problem is that the OT codes must invoke the physical operators yet the OT codes should not need to deal with all the data and parameters needed by the physical operators.

In other words, if a practitioner decides to swap one solver for another, the only thing needed is the name of the new solver.

The operator entrance is for the geophysicist, who formulates the estimation problem. The solver entrance is for the specialist in numerical algebra, who designs a new optimization method. The Fortran-90 programming language allows us to

achieve this design goal by means of generic function interfaces.

A basic solver is `solver_tiny()`. solver_tiny

The two most important arguments in `solver_tiny()` are the operator function `Fop`, which is defined by the interface from Chapter 1, and the stepper function `stepper`, which implements one step of an iterative estimation. For example, a practitioner who choses to use our new `cgstep()` /prog:cgstep for iterative solving the operator `matmult` /prog:matmult would write the call

```
call solver_tiny ( matmult_lop, cgstep, ...
```

so while you are reading the `solver_tiny` module, you should visualize the `Fop()` function as being `matmult_lop`, and you should visualize the `stepper()` function as being `cgstep`.

The other required parameters to `solver_tiny()` are `d` (the data we want to fit), `m` (the model we want to estimate), and `niter` (the maximum number of iterations). There are also a couple of optional arguments. For example, `m0` is the starting guess for the model. If this parameter is omitted, the model is initialized to zero. To output the final residual vector, we include a parameter called `resd`, which is optional as well. We will watch how the list of optional parameters to the generic solver routine grows as we attack more and more complex problems in later chapters.

```
module solver_tiny_mod {                          # 0 = F m - d
contains
  subroutine solver_tiny( m,d, Fop, stepper, niter, m0,resd) {
    optional ::                                    m0,resd
    interface { #--------------------begin definitions -------
       integer function Fop(adj,add,m,d){
         real,dimension(:) :: m,d
         logical,intent(in):: adj,add    }
       integer function stepper(forget,m,g,rr,gg) {
         real,dimension(:) ::          m,g,rr,gg
         logical           ::   forget            }
    }
    real, dimension(:),    intent(in)  :: d       # data
    real, dimension(:),    intent(out) :: m       # model
    real, dimension(:),    intent(in)  :: m0      # initial model
    integer,               intent(in)  :: niter   # number of iterations
    integer                            :: iter    # iteration number
    real, dimension(size( m))          :: g       # gradient (dm)
    real, dimension(size( d)),target   :: rr      # data residual ( vector)
    real, dimension(:),       pointer  :: rd      # data residual (pointer)
    real, dimension(size( d)),target   :: gg      # conj grad     ( vector)
    real, dimension(:),       pointer  :: gd      # conj grad     (pointer)
    integer                            :: stat    # status flag
    real, dimension(:),    intent(out) :: resd    # residual
    rd => rr(1:size( d))
    gd => gg(1:size( d))
    #-------------------------begin initialization -----------
    rd = -d                                  # Rd =      - d
    m  = 0; if(present( m0)){ m = m0         #  m =        m0
        stat = Fop(.false.,.true.,m,rd)      # Rd = Rd + F m0
    }
    do iter = 1,niter { #--------- begin iterations -----------
       stat = Fop( .true.,.false.,g,rd)      #  g =  F' Rd
       stat = Fop(.false.,.false.,g,gd)      # Gd =  F  g
       stat = stepper(.false., m,g, rr,gg)# m = m+dm; R = R + dR
    }
    if(present( resd)) resd = rd
  }
}
```

2.3.8. Why Fortran 90 is much better than Fortran 77

I'd like to digress from our geophysics-mathematics themes to explain why Fortran 90 has been a great step forward over Fortran 77. All the illustrations in this book were originally computed in F77. Then module `solver_tiny` /prog:solver_tiny was simply a subroutine. It was not one module for the whole book, as it is now, but it was many conceptually identical subroutines, dozens of them, one subroutine for each application. The reason for the proliferation was that F77 lacks the ability of F90 to represent operators as having two ways to enter, one for science and another for math. On the other hand, F77 did not require the half a page of definitions that we see here in F90. But the definitions are not difficult to understand, and they are a clutter that we must see once and never again. Another benefit is that the book in F77 had no easy way to switch from the `cgstep` solver to other solvers.

2.3.9. Test case: solving some simultaneous equations

Now we assemble a module `cgmeth` for solving simultaneous equations. Starting with the conjugate-direction module `cgstep_mod` /prog:cgstep we insert the module `matmult` /prog:matmult as the linear operator. cgmeth

```
module cgmeth {
  use matmult
  use cgstep_mod
  use solver_tiny_mod
contains
  # setup of conjugate gradient descent, minimize  SUM rr(i)**2
  #              nx
  # rr(i)  =   sum fff(i,j) * x(j)  -  yy(i)
  #             j=1
  subroutine cgtest( x, yy, rr, fff, niter) {
    real, dimension (:), intent (out) :: x, rr
    real, dimension (:), intent (in)  :: yy
    real, dimension (:,:), pointer    :: fff
    integer,               intent (in)  :: niter
    call matmult_init( fff)
    call solver_tiny( m=x, d=yy, &
        Fop=matmult_lop, stepper=cgstep, &
        niter=niter, resd=rr)
    call cgstep_close ()
  }
}
```

Back

Below shows the solution to 5×4 set of simultaneous equations. Observe that the "exact" solution is obtained in the last step. Because the data and answers are integers, it is quick to check the result manually.

```
d transpose
      3.00      3.00      5.00      7.00      9.00
F transpose
      1.00      1.00      1.00      1.00      1.00
      1.00      2.00      3.00      4.00      5.00
      1.00      0.00      1.00      0.00      1.00
      0.00      0.00      0.00      1.00      1.00
for iter = 0, 4
x    0.43457383  1.56124675  0.27362058  0.25752524
res -0.73055887  0.55706739  0.39193487 -0.06291389 -0.22804642
x    0.51313990  1.38677299  0.87905121  0.56870615
res -0.22103602  0.28668585  0.55251014 -0.37106210 -0.10523783
x    0.39144871  1.24044561  1.08974111  1.46199656
res -0.27836466 -0.12766013  0.20252672 -0.18477242  0.14541438
x    1.00001287  1.00004792  1.00000811  2.00000739
```

```
res  0.00006878  0.00010860  0.00016473  0.00021179  0.00026788
x    1.00000024  0.99999994  0.99999994  2.00000024
res -0.00000001 -0.00000001  0.00000001  0.00000002 -0.00000001
```

EXERCISES:

1 One way to remove a mean value m from signal $s(t) = \mathbf{s}$ is with the fitting goal $\mathbf{0} \approx \mathbf{s} - m$. What operator matrix is involved?

2 What linear operator subroutine from Chapter 1 can be used for finding the mean?

3 How many CD iterations should be required to get the exact mean value?

4 Write a mathematical expression for finding the mean by the CG method.

2.4. INVERSE NMO STACK

To illustrate an example of solving a huge set of simultaneous equations without ever writing down the matrix of coefficients we consider how ***back projection*** can be upgraded towards ***inversion*** in the application called **moveout and stack**.

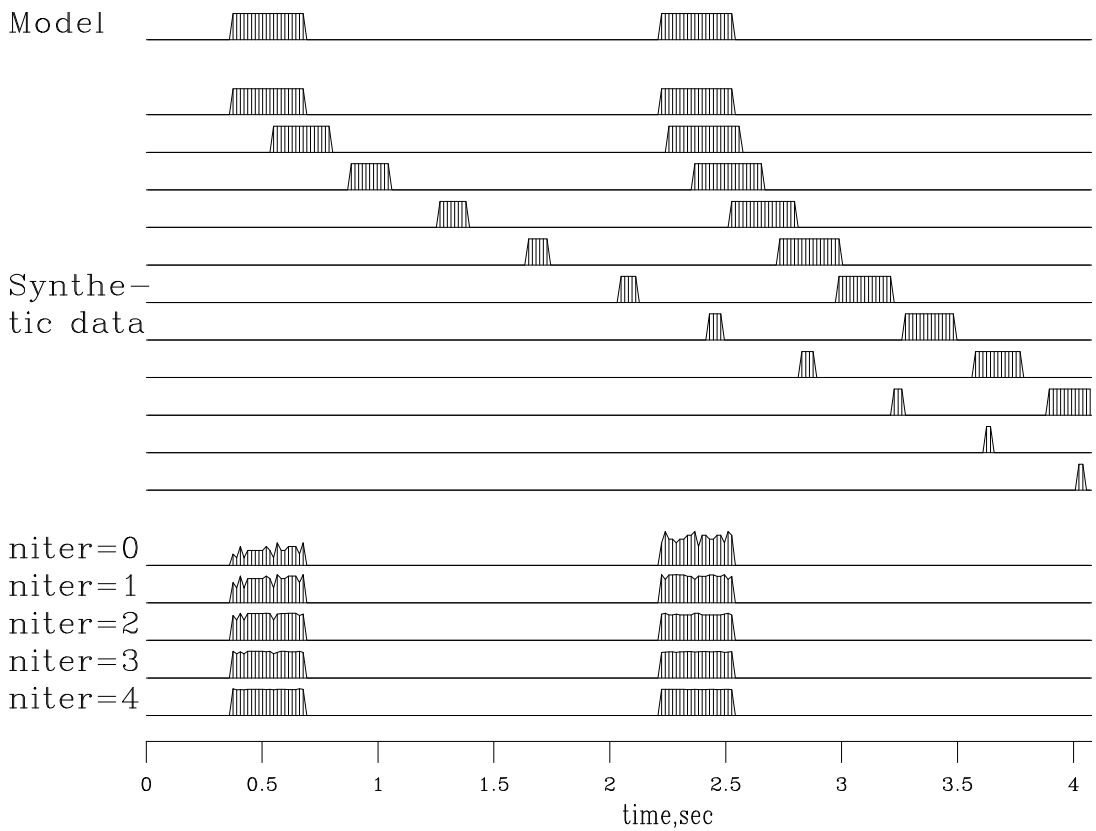


Figure 2.7: Top is a model trace **m**. Next are the synthetic data traces, $\mathbf{d} = \mathbf{Mm}$. Then, labeled `niter=0` is the **stack**, a result of processing by adjoint modeling. Increasing values of `niter` show **x** as a function of iteration count in the fitting goal $\mathbf{d} \approx \mathbf{Mm}$. (Carlos Cunha-Filho) lsq-invstack90 [ER]

```
module invstack {
        use imospray
        use cgstep_mod
        use solver_tiny_mod
contains
        # NMO stack by inverse of forward modeling
        subroutine stack( nt, model, nx, gather,     t0,x0,dt,dx,slow, niter) {
        integer           nt,        nx,                                 niter
        real                  model (:), gather (:), t0,x0,dt,dx,slow
        call imospray_init( slow, x0,dx, t0,dt, nt, nx)
        call solver_tiny( m=model, d=gather, Fop=imospray_lop, step-
per=cgstep, niter=niter)
        call cgstep_close ();  call imospray_close ()   # garbage collection
        }
}
```

Back

The seismograms at the bottom of Figure 2.7 show the first four iterations of conjugate-direction inversion. You see the original rectangle-shaped waveform returning as the iterations proceed. Notice also on the **stack** that the early and late events have unequal amplitudes, but after enough iterations they are equal, as they began. Mathematically, we can denote the top trace as the model $\mathbf{m}$, the synthetic data signals as $\mathbf{d} = \mathbf{M}\mathbf{m}$, and the stack as $\mathbf{M}'\mathbf{d}$. The conjugate-gradient algorithm optimizes the fitting goal $\mathbf{d} \approx \mathbf{M}\mathbf{x}$ by variation of $\mathbf{x}$, and the figure shows $\mathbf{x}$ converging to $\mathbf{m}$. Because there are 256 unknowns in $\mathbf{m}$, it is gratifying to see good convergence occurring after the first four iterations. The fitting is done by module `invstack`, which is just like `cgmeth` /prog:cgmeth except that the matrix-multiplication operator `matmult` /prog:matmult has been replaced by `imospray` /prog:imospray. Studying the program, you can deduce that, except for a scale factor, the output at `niter=0` is identical to the stack $\mathbf{M}'\mathbf{d}$. All the signals in Figure 2.7 are intrinsically the same scale. invstack

This simple inversion is inexpensive. Has anything been gained over conventional stack? First, though we used **nearest-neighbor** interpolation, we managed to preserve the spectrum of the input, apparently all the way to the Nyquist frequency. Second, we preserved the true amplitude scale without ever bothering to think about

(1) dividing by the number of contributing traces, (2) the amplitude effect of NMO stretch, or (3) event truncation.

With depth-dependent velocity, wave fields become much more complex at wide offset. NMO soon fails, but wave-equation forward modeling offers interesting opportunities for inversion.

2.5. FLATTENING 3-D SEISMIC DATA

Here we follow the doctoral disertation of Jesse Lomask. In Figure 2.6 we have seen how to flatten 2-D seismic data. The 3-D process is much more interesting. To see why, consider this: Starting from the origin $(x, y) = (0, 0)$ we move along the x-axis flattening until we come to $(10, 0)$. From there we move along the y-axis flattening until we get to $(x, y) = (10, 10)$. Move backwards on the x-axis to $(0, 10)$ flattening as you go. Finally, return to the origin. Along our journey around this square we have integrated $p = dt/dx$ (and dt/dy) to find the total time shift. Upon returning to the starting point, we would like the total time shift to return to zero. Dealing with real data of less than perfect coherence this might not happen. Old time seismologists would say, "The survey lines don't tie." As we push to the limits

of our knowledge (which we normally do) this problem always arises. We would like a solution that gives the best fit of all the data in a volume. Given a volume of data $u(t,x,y)$ we seek the best $\tau(x,y)$ such that $w(t,x,y) = u(t-\tau(x,y),x,y)$ is flattened. Let's get it.

Here is an expression that on first sight seems to say nothing

$$\nabla\tau \quad = \quad \begin{bmatrix} \frac{\partial\tau}{\partial x} \\ \\ \frac{\partial\tau}{\partial y} \end{bmatrix} \tag{2.81}$$

Equation (2.81) looks like a tautology, a restatement of basic mathematical notation. This is so, however, only if $\tau(x,y)$ is known and the derivatives are derived from it. When $\tau(x,y)$ is not known but the partial derivatives are observed, then we have two measurements at each (x,y) location for the one unknown τ at that location. The same is true at all locations, so we write it as a regression, a residual $\mathbf{r}$ that we will work to get small to find a best fitting $\tau(x,y)$ or maybe $\tau(x,y,t)$. Let $\mathbf{d}$ be the measurements in the vector in equation (2.81), the measurements throughout the (t,x,y)-volume. Expressed as a regression equation (2.81) becomes

$$\mathbf{0} \qu\approx \quad \mathbf{r} \quad = \quad \nabla\tau \quad - \quad \mathbf{d} \tag{2.82}$$

```
module igrad2 {                         # 2-D gradient with adjoint,  r= grad( p)
integer :: n1, n2
#%_init   (n1, n2)
#%_lop ( p(n1, n2),  r(n1,n2,2))
integer i,j
do i= 1, n1-1 {
do j= 1, n2-1 {
        if( adj) {
                p(i+1,j  )  +=  r(i,j,1)
                p(i  ,j  )  -=  r(i,j,1)
                p(i  ,j+1)  +=  r(i,j,2)
                p(i  ,j  )  -=  r(i,j,2)
                }
        else {  r(i,j,1)  +=  ( p(i+1,j) - p(i,j))
                r(i,j,2)  +=  ( p(i,j+1) - p(i,j))
                }
        }}
}
```

Back

Let us see how the coming 3-D illustrations were created. First we need code for vector gradient with its adjoint, negative vector divergence. Here it is: igrad2
In a kind of magic, all we need to fit our regression (2.81) is to pass the `igrad2` module to the Krylov subspace solver, simple solver using `cgstep`, but first we need to compute **d** by calculating dt/dx and dt/dy between all the mesh points.

```
do iy=1,ny { # Calculate x-direction dips: px
  call puck2d(dat(:,:,iy),coh_x,px,res_x,boxsz,nt,nx)
       }
do ix=1,nx { # Calculate y-direction dips: py
  call puck2d(dat(:,ix,:),coh_y,py,res_y,boxsz,nt,ny)
       }
do it=1,nt { # Integrate dips: tau
  call dipinteg(px(it,:,:),py(it,:,:),tau,niter,verb,nx,ny)
       }
```

Finally, initialize the gradient operator, pack the two column vectors dt/dx and dt/dy into a single column vector **d** like in equation (2.81), and tell the simple solver to make its steps with to use `cgstep` with the linear operator `igrad2`. dipinteg

```
module dipinteg_mod{
  use igrad2
  use solver_smp_mod
  use cgstep_mod
  # simple dip integrator for flattening using cg
contains
  subroutine dipinteg(px,py,tau,niter,verb,nx,ny){
    integer, intent(in)             :: niter,nx,ny
    logical, intent(in)             :: verb
    real, dimension(:,:),intent(in ):: px,py
    real, dimension(:),  intent(out):: tau
    real, dimension(:)              :: px_py(2*nx*ny)
    call igrad2_init(nx,ny)
    px_py(1:nx*ny)        =reshape(py,(/nx*ny/))
    px_py(nx*ny+1:2*nx*ny)=reshape(px,(/nx*ny/))
    call solver_smp(tau,px_py,igrad2_lop,cgstep,niter,verb)
  }
}
```

Back

2.5.1. Gulf of Mexico Salt Piercement Example (Jesse Lomask)

Figure 2.8 shows a 3D seismic data cube from the Gulf of Mexico provided by Chevron. A volume of data cannot be displayed on the page of a book. The display here consists of three slices from the volume. Top is a (t_0, x, y) slice, also called a "time slice." Beneath it is a (t, x, y_0) slice; aside that is a (t, x_0, y) slice, depth slices in orthogonal directions. Intersections of the slices within the cube are shown by the heavy black lines on the faces of the cube. The circle in the lower right corner of the top slice is an eruption of salt (which, like ice, under high pressure will flow like a liquid). Inside the salt there are no reflections so the data should be ignored there. Outside the salt we see layers, simple horizons of sedimentary rock. As the salt has pushed upward it has dragged bedding planes upward with it. Presuming the bedding to contain permeable sandstones and impermeable shales, the pushed up bedding around the salt is a prospective oil trap. The time slice in the top panel shows ancient river channels, some large, some small, that are now deeply buried. These may also contain sand. Being natural underground "oil pipes" they are of great interest. To see these pipes as they approach the salt dome we need a picture,

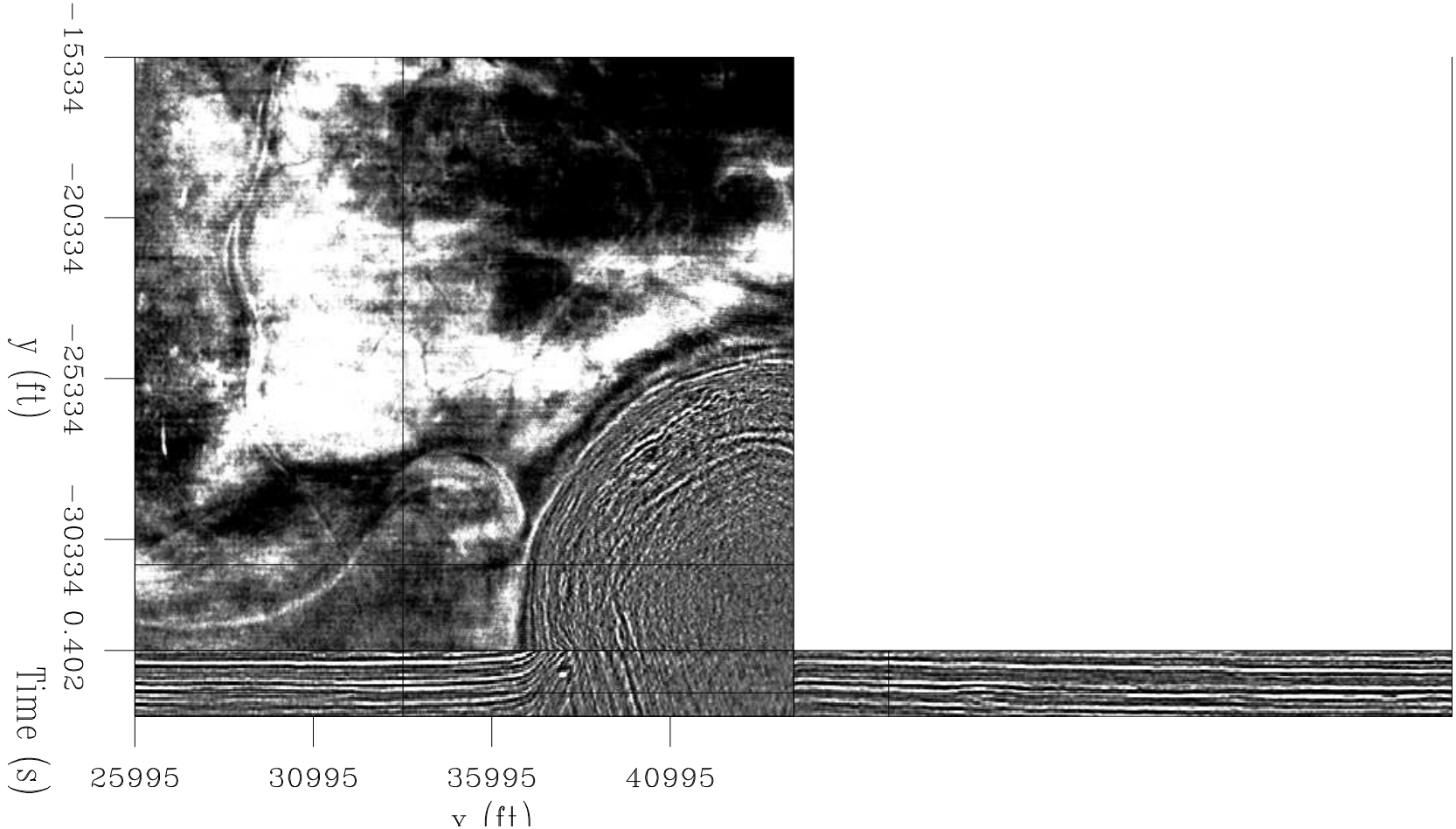


Figure 2.8: Chevron data cube from the Gulf of Mexico. A salt dome (lower left corner in the top plane) has pushed upwards, dragging bedding planes (seen in the bottom two orthogonal planes) along with it. lsq-chev [ER]

not at a constant t, but at a constant $t - \tau(x, y)$.

Figure 2.9 shows a time slice of the original cube and the flattened cube of Figure 2.8. The first thing to notice on the plane before flattening is that the panel drifts from dark to light in place to place. This is because the horizontal layers are not fully horizontal. Approaching the dome the change from dark to light and back again happens so rapidly that the dome appears surrounded by rings. After flattening, the drift and rings disappear. The reflection horizons are no longer cutting across the image. Channels no longer drift off (above or below) the viewable time slice. Carefully viewing the salt dome it seems smaller after flattening because the rings are replace by a bedding plane.

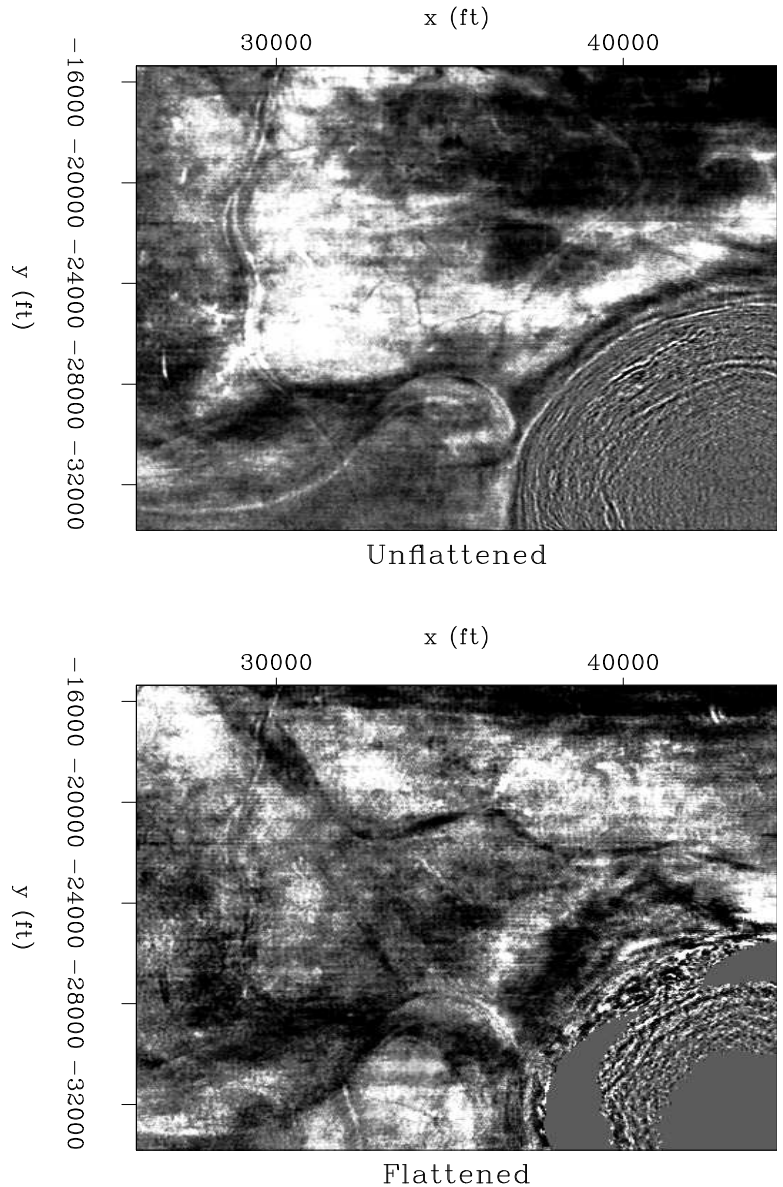


Figure 2.9: Slices of constant time before and after flattening. Notice the rings surrounding the dome are gone giving the dome a reduced diameter. (Ignore the inside of the dome.) lsq-slicecomp22 [ER,M]

2.6. VESUVIUS PHASE UNWRAPPING

Figure 2.10 shows radar[1] images of Mt. Vesuvius[2] in Italy. These images are made from backscatter signals $s_1(t)$ and $s_2(t)$, recorded along two **satellite orbit**s 800 km high and 54 m apart. The signals are very high frequency (the radar wavelength being 2.7 cm). They were Fourier transformed and one multiplied by the complex conjugate of the other, getting the product $Z = S_1(\omega)S_2(\omega)$. The product's amplitude and phase are shown in Figure 2.10. Examining the data, you can notice that where the signals are strongest (darkest on the left), the phase (on the right) is the most spatially consistent. Pixel by pixel evaluation with the two frames in a movie program shows that there are a few somewhat large local amplitudes (clipped in Figure 2.10) but because these generally have spatially consistent phase, I would not describe the data as containing noise bursts.

---

[1] Here we do not require knowledge of radar fundamentals. Common theory and practice is briefly surveyed in Reviews of Geophysics, Vol 36, No 4, November 1998, Radar Interferometry and its application to changes in the earth's surface, Didier Massonnet and Kurt Feigl.

[2] A web search engine quickly finds you other views.

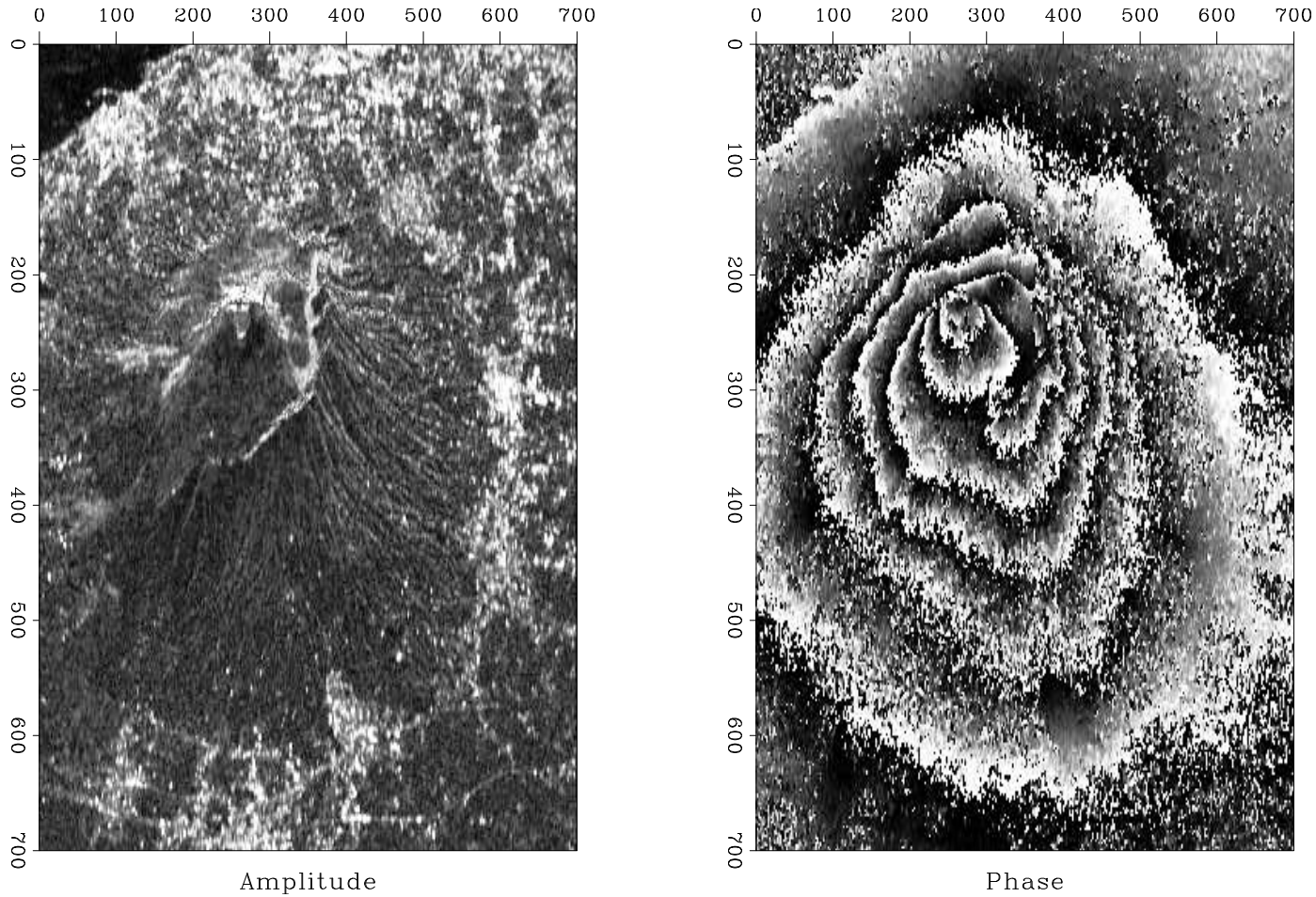


Figure 2.10: Radar image of Mt. Vesuvius. Left is the amplitude. Non-reflecting ocean in upper left corner. Right is the phase. (Umberto Spagnolini) lsq-vesuvio90 [ER,M]

To reduce the time needed for analysis and printing, I reduced the data size two different ways, by decimation and by local averaging, as shown in Figure 2.11. The decimation was to about 1 part in 9 on each axis, and the local averaging was done in 9×9 windows giving the same spatial resolution in each case. The local averaging was done independently in the plane of the real part and the plane of the imaginary part. Putting them back together again showed that the phase angle of the averaged data behaves much more consistently. This adds evidence that the data is not troubled by noise bursts.

From Figures 2.10 and 2.11 we see that **contour**s of constant phase appear to be contours of constant altitude; this conclusion leads us to suppose that a study of radar theory would lead us to a relation like $Z = e^{ih}$ where h is altitude (in units unknown to us nonspecialists). Because the flat land away from the mountain is all at the same phase (as is the altitude), the distance as revealed by the phase does not represent the distance from the ground to the satellite viewer. We are accustomed to measuring altitude along a vertical line to a datum, but here the distance seems to be measured from the ground along a 23° angle from the vertical to a datum at the satellite height.

Phase is a troublesome measurement because we generally see it modulo 2π.

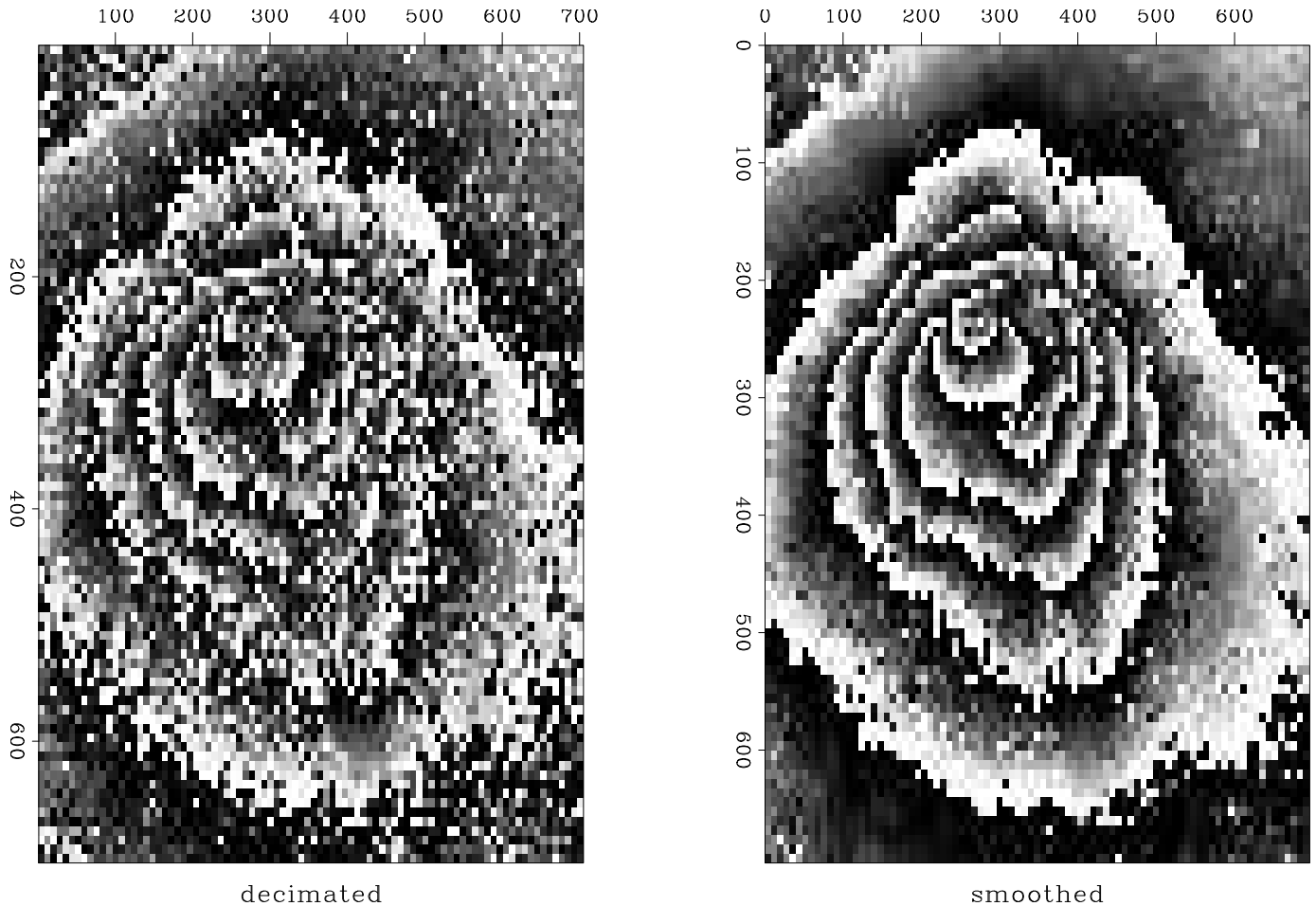


Figure 2.11: Phase based on decimated data (left) and smoothed data (right). lsq-squeeze90 [ER,M]

Marching up the mountain we see the phase getting lighter and lighter until it suddenly jumps to black which then continues to lighten as we continue up the mountain to the next jump. Let us undertake to compute the phase including all of its jumps of 2π. Begin with a complex number Z representing the complex-valued image at any location in the (x, y)-plane.

$$re^{i\phi} = Z \tag{2.83}$$

$$\ln|r| + i(\phi + 2\pi N) = \ln Z \tag{2.84}$$

$$\phi = \Im \ln Z - 2\pi N \tag{2.85}$$

A computer will find the imaginary part of the logarithm with the arctan function of two arguments, `atan2(y,x)`, which will put the phase in the range $-\pi < \phi \leq \pi$ although any multiple of 2π could be added. We seem to escape the $2\pi N$ phase ambiguity by differentiating:

$$\frac{\partial\phi}{\partial x} = \Im \frac{1}{Z}\frac{\partial Z}{\partial x} \tag{2.86}$$

$$\frac{\partial\phi}{\partial x} = \frac{\Im \bar{Z}\frac{\partial Z}{\partial x}}{\bar{Z}Z} \tag{2.87}$$

For every point on the y-axis, equation (2.87) is a differential equation on the x-axis, and we could integrate them all to find $\phi(x,y)$. That sounds easy. On the other hand, the same equations are valid when x and y are interchanged, so we get twice as many equations as unknowns. For ideal data, either of these sets of equations should be equivalent to the other, but for real data we expect to be fitting the fitting goal

$$\nabla\phi \quad \approx \quad \frac{\Im\ \bar{Z}\nabla Z}{\bar{Z}Z} \tag{2.88}$$

where $\nabla = (\frac{\partial}{\partial x}, \frac{\partial}{\partial y})$. This is essentially the same problem we solved flattening seismic data with the regression $\nabla\tau \approx \mathbf{d}$. Taking measurements to be phase differences between neighboring mesh points, it is more correct to interpret equation (2.88) as a difference equation than a differential equation. Since we measure phase differences only over tiny distances (one pixel) we hope not to worry about phases greater than 2π. But if such jumps do occur, they will contribute to overall error.

Let us consider a typical location in the (x,y) plane where the complex numbers

$Z_{i,j}$ are given. Define a shorthand a, b, c, and d as follows:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} Z_{i,j} & Z_{i,j+1} \\ Z_{i+1,j} & Z_{i+1,j+1} \end{bmatrix} \tag{2.89}$$

With this shorthand, the difference equation representation of the fitting goal (2.88) is:

$$\begin{aligned} \phi_{i+1,j} - \phi_{i,j} &\approx \Delta\phi_{ac} \\ \phi_{i,j+1} - \phi_{i,j} &\approx \Delta\phi_{ab} \end{aligned} \tag{2.90}$$

Now let us find the phase jumps between the various locations. Complex numbers a and b may be expressed in polar form, say $a = r_a e^{i\phi_a}$ and $b = r_b e^{i\phi_b}$. The complex number $\bar{a}b = r_a r_b e^{i(\phi_b - \phi_a)}$ has the desired phase $\Delta\phi_{ab}$. To obtain it we take the imaginary part of the complex logarithm $\ln|r_a r_b| + i\,\Delta\phi_{ab}$.

$$\begin{aligned} \phi_b - \phi_a &= \Delta\phi_{ab} = \Im \ln \bar{a}b \\ \phi_d - \phi_c &= \Delta\phi_{cd} = \Im \ln \bar{c}d \\ \phi_c - \phi_a &= \Delta\phi_{ac} = \Im \ln \bar{a}c \\ \phi_d - \phi_b &= \Delta\phi_{bd} = \Im \ln \bar{b}d \end{aligned} \tag{2.91}$$

which gives the information needed to fill in the right-hand side of (2.90), as done by subroutine `igrad2init()` from module `unwrap` /prog:unwrap.

2.6.1. Estimating the inverse gradient

To optimize the fitting goal (2.90), module `unwrap()` uses the conjugate-direction method like the modules `cgmeth()` /prog:cgmeth and `invstack()` /prog:invstack.

unwrap An open question is whether the required number of iterations is reasonable or whether we would need to uncover a preconditioner or more rapid solution method. I adjusted the frame size (by the amount of smoothing in Figure 2.11) so that I would get the solution in about ten seconds with 400 iterations. Results are shown in Figure 2.12. To summarize, the input is the phase map Figure 2.10 and the output is the altitude map in Figure 2.12. Oddly, the input looks maybe nicer than the output because it already looks something like a contour plot. So if we have a beauty contest, then the input beats the output, but if you need to have the (normalized) altitude $h(x,y)$, not the phase of $e^{ih(x,y)}$, then you need to solve the least squares problem.

```
module unwrap {
        use cgstep_mod
        use igrad2
        use solver_smp_mod
contains
        subroutine grad2init( z, n1,n2, rt ) {
        integer  i, j,           n1,n2
        real                rt( n1,n2,2)
        complex              z( n1,n2  ),            a,b,c
        rt = 0.
        do i= 1, n1-1 {
        do j= 1, n2-1 {
                a =  z(i  ,j  )
                c =  z(i+1,j  );     rt(i,j,1) =  aimag( clog( c * conjg( a) ) )
                b =  z(i,  j+1);     rt(i,j,2) =  aimag( clog( b * conjg( a) ) )
                }}
        }
        # Phase unwraper.   Starting from phase hh, improve it.
        subroutine unwraper( zz, hh, niter) {
        integer  n1,n2,              niter
        complex              zz(:,:)
        real                 hh(:)
        real, allocatable ::  rt(:)
        n1 = size( zz, 1)
        n2 = size( zz, 2)
        allocate( rt( n1*n2*2))
        call grad2init( zz,n1,n2, rt)
        call igrad2_init(  n1,n2)
        call solver_smp( m=hh, d=rt, Fop=igrad2_lop, step-
per=cgstep, niter=niter, m0=hh)
        call cgstep_close ()
        deallocate( rt)
        }
}
```

Back

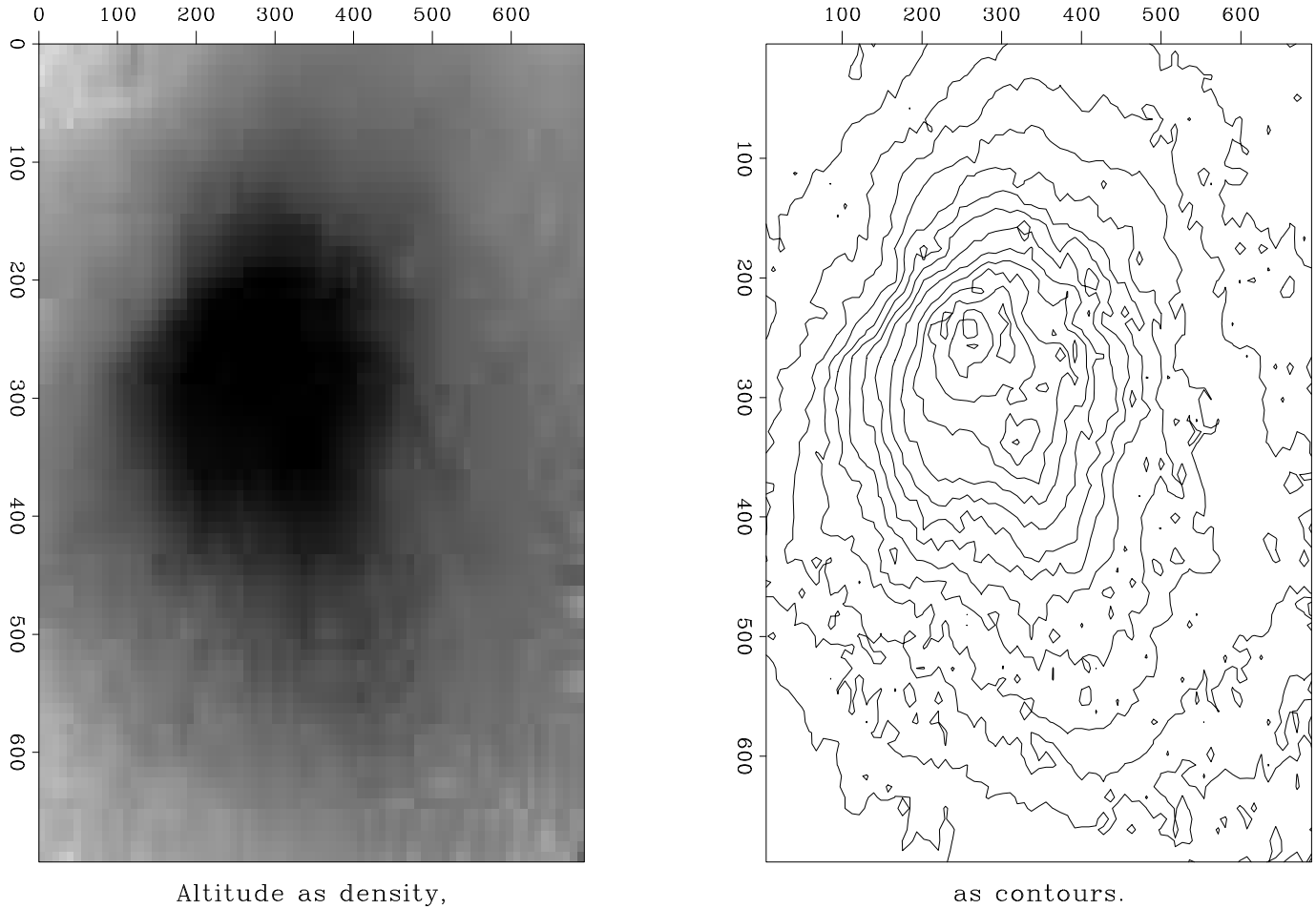


Figure 2.12: Estimated altitude. lsq-veshigh90 [ER,M]

2.6.2. Digression: curl grad as a measure of bad data

The relation (2.91) between the phases and the phase differences is

$$\begin{bmatrix} -1 & 1 & 0 & 0 \\ 0 & 0 & -1 & 1 \\ -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix} \begin{bmatrix} \phi_a \\ \phi_b \\ \phi_c \\ \phi_d \end{bmatrix} = \begin{bmatrix} \Delta\phi_{ab} \\ \Delta\phi_{cd} \\ \Delta\phi_{ac} \\ \Delta\phi_{bd} \end{bmatrix} \tag{2.92}$$

Starting from the phase differences, equation (2.92) cannot find all the phases themselves because an additive constant cannot be found. In other words, the column vector $[1,1,1,1]'$ is in the null space. Likewise, if we add phase increments while we move around a loop, the sum should be zero. Let the loop be $a \to c \to d \to b \to a$. The phase increments that sum to zero are:

$$\Delta\phi_{ac} + \Delta\phi_{cd} - \Delta\phi_{bd} - \Delta\phi_{ab} = 0 \tag{2.93}$$

Rearranging to agree with the order in equation (2.92) yields

$$-\Delta\phi_{ab} + \Delta\phi_{cd} + \Delta\phi_{ac} - \Delta\phi_{bd} = 0 \tag{2.94}$$

which says that the row vector $[-1,+1,+1,-1]$ premultiplies (2.92), yielding zero. Rearrange again

$$-\Delta\phi_{bd} + \Delta\phi_{ac} \quad = \quad \Delta\phi_{ab} - \Delta\phi_{cd} \tag{2.95}$$

and finally interchange signs and directions (i.e., $\Delta\phi_{db} = -\Delta\phi_{bd}$)

$$(\Delta\phi_{db} - \Delta\phi_{ca}) \; - \; (\Delta\phi_{dc} - \Delta\phi_{ba}) \quad = \quad 0 \tag{2.96}$$

This is the finite-difference equivalent of

$$\frac{\partial^2\phi}{\partial x\,\partial y} \; - \; \frac{\partial^2\phi}{\partial y\,\partial x} \quad = \quad 0 \tag{2.97}$$

and is also the z-component of the theorem that the curl of a gradient $\nabla \times \nabla\phi$ vanishes for any ϕ.

The four $\Delta\phi$ summed around the 2×2 mesh should add to zero. I wondered what would happen if random complex numbers were used for a, b, c, and d, so I computed the four $\Delta\phi$'s with equation (2.91), and then computed the sum with (2.93). They did sum to zero for 2/3 of my random numbers. Otherwise, with probability 1/6 each, they summed to $\pm 2\pi$. The nonvanishing curl represents a phase

that is changing too rapidly between the mesh points. Figure 2.13 shows the locations at Vesuvius where bad data occurs. This is shown at two different resolutions. The figure shows a tendency for bad points with curl 2π to have a neighbor with -2π. If Vesuvius were random noise instead of good data, the planes in Figure 2.13 would be one-third covered with dots but as expected, we see considerably fewer.

2.6.3. Discontinuity in the solution

The viewing angle (23 degrees off vertical) in Figure 2.10 might be such that the mountain blocks some of the landscape behind it. This leads to the interesting possibility that the phase function must have a discontinuity where our viewing angle jumps over the hidden terrain. It will be interesting to discover whether we can estimate functions with such discontinuities. I am not certain that the Vesuvius data really has such a shadow zone, so I prepared the synthetic data in Figure 2.14, which is noise free and definitely has one.

We notice the polarity of the synthetic data in 2.14 is opposite that of the Vesuvius data. This means that the radar altitude of Vesuvius is not measured from sea level but from the satellite level.

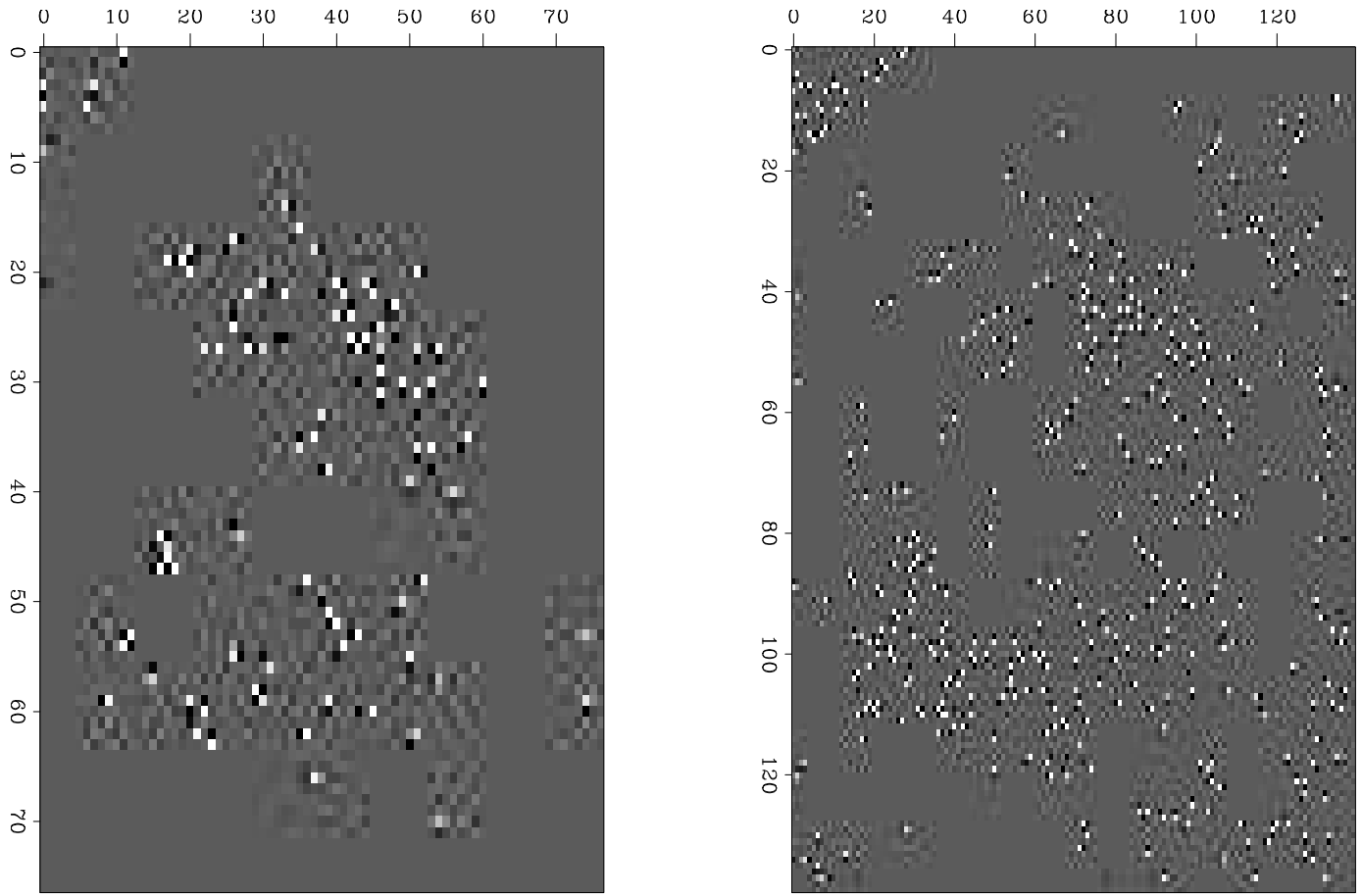


Figure 2.13: Values of curl at Vesuvius. The bad data locations at both coarse and fine resolution tend to occur in pairs of opposite polarity. lsq-screw90 [ER,M]

EXERCISES:

1 In differential equations, boundary conditions are often (1) a specified function value or (2) a specified derivative. These are associated with (1) transient convolution or (2) internal convolution. Gradient operator `igrad2` /prog:igrad2 is based on internal convolution with the filter $(1, -1)$. Revise `igrad2` to make a module called `tgrad2` which has transient boundaries.

Here is a real-world problem you could think about: You have earthquake seismograms recorded at $i = 1, 2, ..., N$ locations not on a regular mesh. You would like to shift them into alignment. Assume a cartesian geometry. You have measured all possible time lags $\tau_{i,j}$ between station i and station j. What operator would you be giving to the solver?

2.6.4. Analytical solutions

We have found a numerical solution to fitting problems such as this

$$\mathbf{0} \quad \approx \quad \nabla\tau \quad - \quad \mathbf{d} \tag{2.98}$$

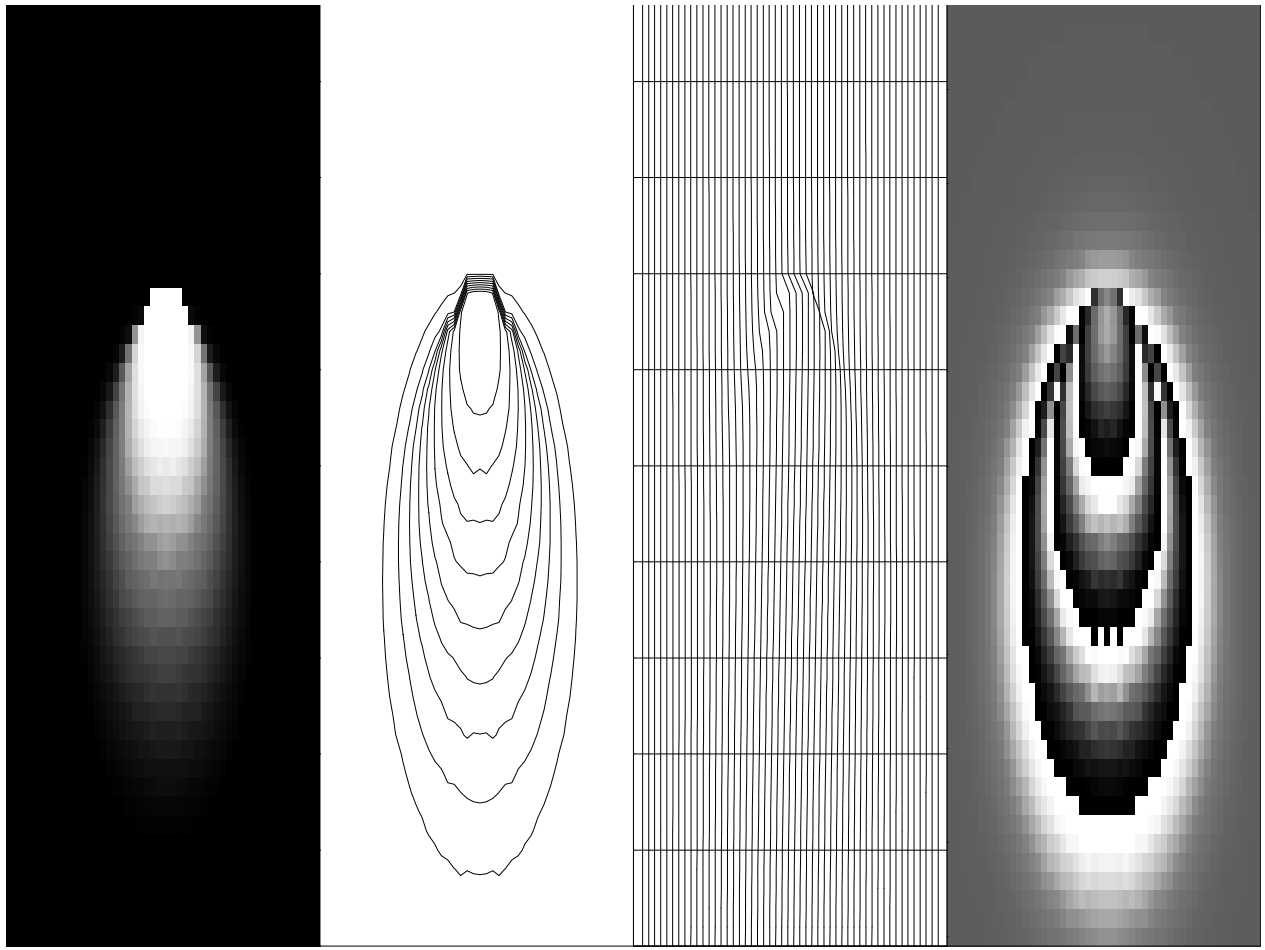


Figure 2.14: Synthetic mountain with hidden backside. For your estimation enjoyment. lsq-synmod90 [ER,M]

An analytical solution will be much faster. From any regression we get the least squares solution when we multipy by the transpose of the operator. Thus

$$\mathbf{0} \quad = \quad \nabla' \nabla \tau \quad - \quad \nabla' \mathbf{d} \tag{2.99}$$

We need to understand what is the transpose of the gradient operator. Recall the finite difference representation of a derivative in chapter 1. Ignoring end effects, the transpose of a derivative is the negative of a derivative. Since the transpose of a column vector is a row vector, the adjoint of a gradient ∇, namely, ∇' is more commonly known as the vector divergence $\nabla \cdot$. Likewise $\nabla' \nabla$ is a positive definite matrix, the negative of the Laplacian ∇^2. Thus, in more conventional mathematical notation, the solution τ is that of Poisson's equation.

$$\nabla^2 \tau \quad = \quad - \nabla \cdot \mathbf{d} \tag{2.100}$$

In the Fourier domain we can have an analytic solution. There $-\nabla^2 = k_x^2 + k_y^2$ where (k_x, k_y) are the Fourier frequencies on the (x, y) axes. Instead of thinking of equation (2.100) as a convolution in physical space, think of it as a product in

Fourier space. Thus, the analytic solution is

$$\tau(x,y) \quad = \quad \mathbf{FT}^{-1} \frac{\mathbf{FT}\ \nabla \cdot \mathbf{d}}{k_x^2 + k_y^2} \tag{2.101}$$

where **FT** denotes 2-dimensional Fourier transform over x and y.

Here is a trick from numerical analysis that gives better results: Instead of representing the denominator $k_x^2 + k_y^2$ in the most obvious way, let us represent it in a manner consistant with the finite-difference way we expressed the numerator $\nabla \cdot \mathbf{d}$. Recall that $-i\omega\Delta t \approx -i\hat{\omega}\Delta t = 1 - Z = 1 - \exp(-i\omega\Delta t)$ which is a Fourier domain way of saying that difference equations tend to differential equations at low frequencies. Likewise a symmetric second time derivative has a finite-difference representation proportional to $(-2 + Z + 1/Z)$ and in a two-dimensional space, a finite-difference representation of the Laplacian operator is proportional to $(-4 + X + 1/X + Y + 1/Y)$ where $X = \exp(ik_x\Delta x)$ and $Y = \exp(ik_y\Delta y)$.

Fourier solutions have their own peculiarities (periodic boundary conditions) which are not always appropriate in practice, but having these solutions available is often a nice place to start from when solving a problem that cannot be solved in Fourier space. For example, suppose we feel some data values are bad and we

would like to throw out the regression equations involving the bad data points. At Vesuvius we might consider the strength of the radar return (which we have previously ignored) and use it as a weighting function $\mathbf{W}$. Now our regression (2.98) becomes

$$\mathbf{0} \quad \approx \quad \mathbf{W}\,(\nabla\phi \;-\; \mathbf{d}) \quad = \quad (\mathbf{W}\,\nabla)\phi \;-\; \mathbf{W}\mathbf{d} \tag{2.102}$$

This is a problem we know how to solve, a regression with an operator $\mathbf{W}\nabla$ and data $\mathbf{W}\mathbf{d}$. The weighted problem is not solveable in the Fourier domain because the operator $(\mathbf{W}\nabla)'\mathbf{W}\nabla$ has no simple expression in the Fourier domain. Thus we would use the analytic solution to the unweighted problem as a starting guess for the iterative solution to the real problem.

With the Vesuvius data we might we construct the weight $\mathbf{W}$ from the signal strength. We also have available the curl, which should vanish. Its non-vanishing is an indicator of questionable data which could be weighted down relative to other data.

2.7. THE WORLD OF CONJUGATE GRADIENTS

Nonlinearity arises in two ways: First, theoretical data might be a nonlinear function of the model parameters. Second, observed data could contain imperfections that force us to use **nonlinear methods** of statistical estimation.

2.7.1. Physical nonlinearity

When standard methods of physics relate theoretical data $\mathbf{d}_{\text{theor}}$ to model parameters $\mathbf{m}$, they often use a nonlinear relation, say $\mathbf{d}_{\text{theor}} = \mathbf{f}(\mathbf{m})$. The power-series approach then leads to representing theoretical data as

$$\mathbf{d}_{\text{theor}} \quad = \quad \mathbf{f}(\mathbf{m}_0 + \Delta\mathbf{m}) \quad \approx \quad \mathbf{f}(\mathbf{m}_0) + \mathbf{F}\Delta\mathbf{m} \tag{2.103}$$

where $\mathbf{F}$ is the matrix of partial derivatives of data values by model parameters, say $\partial d_i / \partial m_j$, evaluated at $\mathbf{m}_0$. The theoretical data $\mathbf{d}_{\text{theor}}$ minus the observed data $\mathbf{d}_{\text{obs}}$ is the residual we minimize.

$$\mathbf{0} \quad \approx \quad \mathbf{d}_{\text{theor}} - \mathbf{d}_{\text{obs}} \quad = \quad \mathbf{F}\Delta\mathbf{m} + [\mathbf{f}(\mathbf{m}_0) - \mathbf{d}_{\text{obs}}] \tag{2.104}$$

$$\mathbf{r}_{\text{new}} \quad = \quad \mathbf{F}\Delta\mathbf{m} + \mathbf{r}_{\text{old}} \tag{2.105}$$

It is worth noticing that the residual updating (2.105) in a nonlinear problem is the same as that in a linear problem (2.57). If you make a large step $\Delta\mathbf{m}$, however, the new residual will be different from that expected by (2.105). Thus you should always re-evaluate the residual vector at the new location, and if you are reasonably cautious, you should be sure the residual norm has actually decreased before you accept a large step.

The pathway of inversion with physical nonlinearity is well developed in the academic literature and Bill **Symes** at Rice University has a particularly active group.

2.7.2. Statistical nonlinearity

The data itself often has **noise bursts** or **gaps**, and we will see later in Chapter 7 that this leads us to readjusting the **weighting function**. In principle, we should fix the weighting function and solve the problem. Then we should revise the weighting function and solve the problem again. In practice we find it convenient to change the weighting function during the optimization descent. Failure is possible when the weighting function is changed too rapidly or drastically. (The proper way to

solve this problem is with robust estimators. Unfortunately, I do not yet have an all-purpose robust solver. Thus we are (temporarily, I hope) reduced to using crude reweighted least-squares methods. Sometimes they work and sometimes they don't.)

2.7.3. Coding nonlinear fitting problems

We can solve nonlinear least-squares problems in about the same way as we do iteratively reweighted ones. A simple adaptation of a linear method gives us a **nonlinear solver** if the residual is recomputed at each iteration. Omitting the weighting function (for simplicity) the **template** is:

iterate {

$\mathbf{r} \quad \longleftarrow \quad \mathbf{f(m)} - \mathbf{d}$

Define $\mathbf{F} = \partial \mathbf{d} / \partial \mathbf{m}$.

$\Delta \mathbf{m} \quad \longleftarrow \quad \mathbf{F}' \, \mathbf{r}$

$\Delta \mathbf{r} \quad \longleftarrow \quad \mathbf{F} \, \Delta \mathbf{m}$

$(\mathbf{m}, \mathbf{r}) \quad \longleftarrow \quad \text{step}(\mathbf{m}, \mathbf{r}, \Delta \mathbf{m}, \Delta \mathbf{r})$

}

A formal theory for the optimization exists, but we are not using it here. The assumption we make is that the step size will be small, so that familiar line-search and plane-search approximations should succeed in reducing the residual. Unfortunately this assumption is not reliable. What we should do is test that the residual really does decrease, and if it does not we should revert to steepest descent with a smaller step size. Perhaps we should test an incremental variation on the status quo: where inside `solver` /prog:solver_tiny, we check to see if the residual diminished in the *previous* step, and if it did not, restart the iteration (choose the *current* step to be steepest descent instead of CD). I am planning to work with some mathematicians to gain experience with other solvers.

Experience shows that nonlinear problems have many pitfalls. Start with a linear problem, add a minor physical improvement or unnormal noise, and the problem becomes nonlinear and probably has another solution far from anything reasonable. When solving such a nonlinear problem, we cannot arbitrarily begin from zero as we do with linear problems. We must choose a reasonable starting guess, and then move in a stable and controlled manner. A simple solution is to begin with several steps of steepest descent and then switch over to do some more steps of CD. Avoiding CD in earlier iterations can avoid instability. Strong linear “regulariza-

tion" discussed later can also reduce the effect of nonlinearity.

2.7.4. Standard methods

The conjugate-direction method is really a family of methods. Mathematically, where there are n unknowns, these algorithms all converge to the answer in n (or fewer) steps. The various methods differ in numerical accuracy, treatment of underdetermined systems, accuracy in treating ill-conditioned systems, space requirements, and numbers of dot products. Technically, the method of CD used in the `cgstep` module /prog:cgstep is not the conjugate-gradient method itself, but is equivalent to it. This method is more properly called the **conjugate-direction method** with a memory of one step. I chose this method for its clarity and flexibility. If you would like a free introduction and summary of conjugate-gradient methods, I particularly recommend *An Introduction to Conjugate Gradient Method Without Agonizing Pain* by Jonathon Shewchuk, which you can download[3].

[3] http://www.cs.cmu.edu/afs/cs/project/quake/public/papers/painless-conjugate-gradient.ps

I suggest you skip over the remainder of this section and return after you have seen many examples and have developed some expertise, and have some technical problems.

The **conjugate-gradient method** was introduced by **Hestenes** and **Stiefel** in 1952. To read the standard literature and relate it to this book, you should first realize that when I write fitting goals like

$$0 \quad \approx \quad \mathbf{W}(\mathbf{F}\mathbf{m} - \mathbf{d}) \tag{2.106}$$

$$0 \quad \approx \quad \mathbf{A}\mathbf{m}, \tag{2.107}$$

they are equivalent to minimizing the quadratic form:

$$\mathbf{m}: \qquad \min_{\mathbf{m}} Q(\mathbf{m}) \quad = \quad (\mathbf{m}'\mathbf{F}' - \mathbf{d}')\mathbf{W}'\mathbf{W}(\mathbf{F}\mathbf{m} - \mathbf{d}) \;+\; \mathbf{m}'\mathbf{A}'\mathbf{A}\mathbf{m} \tag{2.108}$$

The optimization theory (OT) literature starts from a minimization of

$$\mathbf{x}: \qquad \min_{\mathbf{x}} Q(\mathbf{x}) \quad = \quad \mathbf{x}'\mathbf{H}\mathbf{x} - \mathbf{b}'\mathbf{x} \tag{2.109}$$

To relate equation (2.108) to (2.109) we expand the parentheses in (2.108) and abandon the constant term $\mathbf{d}'\mathbf{d}$. Then gather the quadratic term in $\mathbf{m}$ and the linear term in $\mathbf{m}$. There are two terms linear in $\mathbf{m}$ that are transposes of each other. They

are scalars so they are equal. Thus, to invoke "standard methods," you take your problem-formulation operators **F**, **W**, **A** and create two modules that apply the operators

$$\mathbf{H} = \mathbf{F}'\mathbf{W}'\mathbf{W}\mathbf{F} + \mathbf{A}'\mathbf{A} \tag{2.110}$$

$$\mathbf{b}' = 2(\mathbf{F}'\mathbf{W}'\mathbf{W}\mathbf{d})' \tag{2.111}$$

The operators **H** and $\mathbf{b}'$ operate on model space. Standard procedures do not require their adjoints because **H** is its own adjoint and $\mathbf{b}'$ reduces model space to a scalar. You can see that computing **H** and $\mathbf{b}'$ requires one temporary space the size of data space (whereas `cgstep` requires two).

When people have trouble with conjugate gradients or conjugate directions, I always refer them to the **Paige and Saunders algorithm** `LSQR`. Methods that form **H** explicitly or implicitly (including both the standard literature and the book3 method) square the condition number, that is, they are twice as susceptible to rounding error as is `LSQR`.

2.7.5. Understanding CG magic and advanced methods

This section includes Sergey Fomel's explanation on the "magic" convergence properties of the conjugate-direction methods. It also presents a classic version of conjugate gradients, which can be found in numerous books on least-square optimization. The key idea for constructing an optimal iteration is to update the solution at each step in the direction, composed by a linear combination of the current direction and all previous solution steps. To see why this is a helpful idea, let us consider first the method of random directions. Substituting expression (2.60) into formula (2.58), we see that the residual power decreases at each step by

$$\mathbf{r}\cdot\mathbf{r}-\mathbf{r}_{\text{new}}\cdot\mathbf{r}_{\text{new}} \quad = \quad \frac{(\mathbf{r}\cdot\Delta\mathbf{r})^2}{(\Delta\mathbf{r}\cdot\Delta\mathbf{r})}\ . \tag{2.112}$$

To achieve a better convergence, we need to maximize the right hand side of (2.112). Let us define a new solution step $\mathbf{s}_{\text{new}}$ as a combination of the current direction $\Delta\mathbf{x}$ and the previous step $\mathbf{s}$, as follows:

$$\mathbf{s}_{\text{new}} \quad = \quad \Delta\mathbf{x}+\beta\mathbf{s}\ . \tag{2.113}$$

The solution update is then defined as

$$\mathbf{x}_{\text{new}} \quad = \quad \mathbf{x} + \alpha \mathbf{s}_{\text{new}} \; . \tag{2.114}$$

The formula for α (2.60) still holds, because we have preserved in (2.114) the form of equation (2.54) and just replaced $\Delta\mathbf{x}$ with $\mathbf{s}_{\text{new}}$. In fact, formula (2.60) can be simplified a little bit. From (2.59), we know that $\mathbf{r}_{\text{new}}$ is orthogonal to $\Delta\mathbf{r} = \mathbf{F}\mathbf{s}_{\text{new}}$. Likewise, $\mathbf{r}$ should be orthogonal to $\mathbf{F}\mathbf{s}$ (recall that $\mathbf{r}$ was $\mathbf{r}_{\text{new}}$ and $\mathbf{s}$ was $\mathbf{s}_{\text{new}}$ at the previous iteration). We can conclude that

$$(\mathbf{r} \cdot \Delta\mathbf{r}) \quad = \quad (\mathbf{r} \cdot \mathbf{F}\mathbf{s}_{\text{new}}) \quad = \quad (\mathbf{r} \cdot \mathbf{F}\Delta\mathbf{x}) + \beta(\mathbf{r} \cdot \mathbf{F}\mathbf{s}) \quad = \quad (\mathbf{r} \cdot \mathbf{F}\Delta\mathbf{x}) \, . \tag{2.115}$$

Comparing (2.115) with (2.112), we can see that adding a portion of the previous step to the current direction does not change the value of the numerator in expression (2.112). However, the value of the denominator can be changed. Minimizing the denominator maximizes the residual increase at each step and leads to a faster convergence. This is the denominator minimization that constrains the value of the adjustable coefficient β in (2.113).

The procedure for finding β is completely analogous to the derivation of for-

mula (2.60). We start with expanding the dot product $(\Delta\mathbf{r}\cdot\Delta\mathbf{r})$:

$$(\mathbf{Fs}_{\rm new}\cdot\mathbf{Fs}_{\rm new}) \quad = \quad \mathbf{F}\Delta\mathbf{x}\cdot\mathbf{F}\Delta\mathbf{x}+2\beta(\mathbf{F}\Delta\mathbf{x}\cdot\mathbf{Fs})+\beta^2\,\mathbf{Fs}\cdot\mathbf{Fs}\,. \tag{2.116}$$

Differentiating with respect to β and setting the derivative to zero, we find that

$$0 \quad = \quad 2(\mathbf{F}\Delta\mathbf{x}+\beta\mathbf{Fs})\cdot\mathbf{Fs}\,. \tag{2.117}$$

Equation (2.117) states that the *conjugate direction* $\mathbf{Fs}_{\rm new}$ is orthogonal (perpendicular) to the previous conjugate direction $\mathbf{Fs}$. It also defines the value of β as

$$\beta \quad = \quad -\frac{(\mathbf{F}\Delta\mathbf{x}\cdot\mathbf{Fs})}{(\mathbf{Fs}\cdot\mathbf{Fs})}\,. \tag{2.118}$$

Can we do even better? The positive quantity that we minimized in (2.116) decreased by

$$\mathbf{F}\Delta\mathbf{x}\cdot\mathbf{F}\Delta\mathbf{x}-\mathbf{Fs}_{\rm new}\cdot\mathbf{Fs}_{\rm new} \quad = \quad \frac{(\mathbf{F}\Delta\mathbf{x}\cdot\mathbf{Fs})^2}{(\mathbf{Fs}\cdot\mathbf{Fs})} \tag{2.119}$$

Can we decrease it further by adding another previous step? In general, the answer is positive, and it defines the method of conjugate directions. I will state this result without a formal proof (which uses the method of mathematical induction).

- If the new step is composed of the current direction and a combination of all the previous steps:

$$\mathbf{s}_n \quad = \quad \Delta\mathbf{x}_n + \sum_{i<n} \beta_i \mathbf{s}_i \ , \tag{2.120}$$

  then the optimal convergence is achieved when

$$\beta_i \quad = \quad -\frac{(\mathbf{F}\Delta\mathbf{x}_n \cdot \mathbf{F}\mathbf{s}_i)}{(\mathbf{F}\mathbf{s}_i \cdot \mathbf{F}\mathbf{s}_i)} \ . \tag{2.121}$$

- The new conjugate direction is orthogonal to the previous ones:

$$(\mathbf{F}\mathbf{s}_n \cdot \mathbf{F}\mathbf{s}_i) \quad = \quad 0 \quad \text{for all} \quad i < n \tag{2.122}$$

To see why this is an optimally convergent method, it is sufficient to notice that vectors $\mathbf{F}\mathbf{s}_i$ form an orthogonal basis in the data space. The vector from the current residual to the smallest residual also belongs to that space. If the data size is n, then n basis components (at most) are required to represent this vector, hence no more then n conjugate-direction steps are required to find the solution.

The computation template for the method of conjugate directions is

$$
\begin{array}{lcl}
\mathbf{r} & \longleftarrow & \mathbf{F}\mathbf{x}-\mathbf{d} \\
\text{iterate } \{ & & \\
\quad \Delta\mathbf{x} & \longleftarrow & \text{random numbers} \\
\quad \mathbf{s} & \longleftarrow & \Delta\mathbf{x}+\sum_{i<n}\beta_i\mathbf{s}_i \quad \text{where} \quad \beta_i=-\frac{(\mathbf{F}\Delta\mathbf{x}\cdot\mathbf{F}\mathbf{s}_i)}{(\mathbf{F}\mathbf{s}_i\cdot\mathbf{F}\mathbf{s}_i)} \\
\quad \Delta\mathbf{r} & \longleftarrow & \mathbf{F}\mathbf{s} \\
\quad \alpha & \longleftarrow & -(\mathbf{r}\cdot\Delta\mathbf{r})/(\Delta\mathbf{r}\cdot\Delta\mathbf{r}) \\
\quad \mathbf{x} & \longleftarrow & \mathbf{x}+\alpha\mathbf{s} \\
\quad \mathbf{r} & \longleftarrow & \mathbf{r}+\alpha\Delta\mathbf{r} \\
\quad \} & &
\end{array}
$$

What happens if we "feed" the method with gradient directions instead of just random directions? It turns out that in this case we need to remember from all the previous steps $\mathbf{s}_i$ only the one that immediately precedes the current iteration. Let us derive a formal proof of that fact as well as some other useful formulas related to the method of *conjugate gradients*.

According to formula (2.59), the new residual $\mathbf{r}_{\text{new}}$ is orthogonal to the conjugate direction $\Delta\mathbf{r}=\mathbf{F}\mathbf{s}_{\text{new}}$. According to the orthogonality condition (2.122), it is also orthogonal to all the previous conjugate directions. Defining $\Delta\mathbf{x}$ equal to the gradient $\mathbf{F}'\mathbf{r}$ and applying the definition of the adjoint operator, it is convenient to

rewrite the orthogonality condition in the form

$$0 \quad = \quad (\mathbf{r}_n \cdot \mathbf{F}\mathbf{s}_i) \quad = \quad (\mathbf{F}'\mathbf{r}_n \cdot \mathbf{s}_i) \quad = \quad (\Delta\mathbf{x}_{n+1} \cdot \mathbf{s}_i) \quad \text{for all} \quad i \le n \tag{2.123}$$

According to formula (2.120), each solution step $\mathbf{s}_i$ is just a linear combination of the gradient $\Delta\mathbf{x}_i$ and the previous solution steps. We deduce from formula (2.123) that

$$0 \quad = \quad (\Delta\mathbf{x}_n \cdot \mathbf{s}_i) \quad = \quad (\Delta\mathbf{x}_n \cdot \Delta\mathbf{x}_i) \quad \text{for all} \quad i < n \tag{2.124}$$

In other words, in the method of conjugate gradients, the current gradient direction is always orthogonal to all the previous directions. The iteration process constructs not only an orthogonal basis in the data space but also an orthogonal basis in the model space, composed of the gradient directions.

Now let us take a closer look at formula (2.121). Note that $\mathbf{F}\mathbf{s}_i$ is simply related to the residual step at i-th iteration:

$$\mathbf{F}\mathbf{s}_i = \frac{\mathbf{r}_i - \mathbf{r}_{i-1}}{\alpha_i} \; . \tag{2.125}$$

Substituting relationship (2.125) into formula (2.121) and applying again the defi-

nition of the adjoint operator, we obtain

$$\beta_i = -\frac{\mathbf{F}\Delta\mathbf{x}_n \cdot (\mathbf{r}_i - \mathbf{r}_{i-1})}{\alpha_i(\mathbf{F}\mathbf{s}_i \cdot \mathbf{F}\mathbf{s}_i)} = -\frac{\Delta\mathbf{x}_n \cdot \mathbf{F}'(\mathbf{r}_i - \mathbf{r}_{i-1})}{\alpha_i(\mathbf{F}\mathbf{s}_i \cdot \mathbf{F}\mathbf{s}_i)} = -\frac{\Delta\mathbf{x}_n \cdot (\Delta\mathbf{x}_{i+1} - \Delta\mathbf{x}_i)}{\alpha_i(\mathbf{F}\mathbf{s}_i \cdot \mathbf{F}\mathbf{s}_i)} \tag{2.126}$$

Since the gradients $\Delta\mathbf{x}_i$ are orthogonal to each other, the dot product in the numerator is equal to zero unless $i = n-1$. It means that only the immediately preceding step $\mathbf{s}_{n-1}$ contributes to the definition of the new solution direction $\mathbf{s}_n$ in (2.120). This is precisely the property of the conjugate gradient method we wanted to prove.

To simplify formula (2.126), rewrite formula (2.60) as

$$\alpha_i = -\frac{(\mathbf{r}_{i-1} \cdot \mathbf{F}\Delta\mathbf{x}_i)}{(\mathbf{F}\mathbf{s}_i \cdot \mathbf{F}\mathbf{s}_i)} = -\frac{(\mathbf{F}'\mathbf{r}_{i-1} \cdot \Delta\mathbf{x}_i)}{(\mathbf{F}\mathbf{s}_i \cdot \mathbf{F}\mathbf{s}_i)} = -\frac{(\Delta\mathbf{x}_i \cdot \Delta\mathbf{x}_i)}{(\mathbf{F}\mathbf{s}_i \cdot \mathbf{F}\mathbf{s}_i)} \tag{2.127}$$

Substituting (2.127) into (2.126), we obtain

$$\beta = -\frac{(\Delta\mathbf{x}_n \cdot \Delta\mathbf{x}_n)}{\alpha_{n-1}(\mathbf{F}\mathbf{s}_{n-1} \cdot \mathbf{F}\mathbf{s}_{n-1})} = \frac{(\Delta\mathbf{x}_n \cdot \Delta\mathbf{x}_n)}{(\Delta\mathbf{x}_{n-1} \cdot \Delta\mathbf{x}_{n-1})}\,. \tag{2.128}$$

The computation template for the method of conjugate gradients is then

$$
\begin{aligned}
&\mathbf{r} \longleftarrow \mathbf{F}\mathbf{x} - \mathbf{d} \\
&\beta \longleftarrow 0 \\
&\text{iterate } \{ \\
&\qquad \Delta\mathbf{x} \longleftarrow \mathbf{F}'\mathbf{r} \\
&\qquad \text{if not the first iteration } \beta \longleftarrow \frac{(\Delta\mathbf{x}\cdot\Delta\mathbf{x})}{\gamma} \\
&\qquad \gamma \longleftarrow (\Delta\mathbf{x}\cdot\Delta\mathbf{x}) \\
&\qquad \mathbf{s} \longleftarrow \Delta\mathbf{x} + \beta\mathbf{s} \\
&\qquad \Delta\mathbf{r} \longleftarrow \mathbf{F}\mathbf{s} \\
&\qquad \alpha \longleftarrow -\gamma/(\Delta\mathbf{r}\cdot\Delta\mathbf{r}) \\
&\qquad \mathbf{x} \longleftarrow \mathbf{x} + \alpha\mathbf{s} \\
&\qquad \mathbf{r} \longleftarrow \mathbf{r} + \alpha\Delta\mathbf{r} \\
&\qquad \} \\
\end{aligned}
$$

2.8. REFERENCES

Hestenes, M.R., and Stiefel, E., 1952, Methods of conjugate gradients for solving linear systems: J Res. Natl. Bur. Stand., **49**, 409-436.

Paige, C.C., and Saunders, M.A., 1982a, LSQR: an algorithm for sparse linear equations and sparse least squares: Assn. Comp. Mach. Trans. Mathematical Software, **8,** 43-71.

Paige, C.C., and Saunders, M.A., 1982b, Algorithm 583, LSQR: sparse linear equations and least squares problems: Assn. Comp. Mach. Trans. Mathematical Software, **8,** 195-209.

Chapter 3

Empty bins and inverse interpolation

Let us review the big picture. In Chapter 1 we developed adjoints and in Chapter 2 we developed inverse operators. Logically, correct solutions come only through

inversion. Real life, however, seems nearly the opposite. This is puzzling but intriguing.

Every time you fill your car with gasoline, it derives much more from the adjoint than from inversion. I refer to the fact that "practical seismic data processing" relates much more to the use of adjoints than of inverses. It has been widely known for about the last 15 years that medical imaging and all basic image creation methods are like this. It might seem that an easy path to fame and profit would be to introduce the notion of inversion, but it is not that easy. Both cost and result quality enter the picture.

First consider cost. For simplicity, consider a data space with N values and a model (or image) space of the same size. The computational cost of applying a dense adjoint operator increases in direct proportion to the number of elements in the matrix, in this case N^2. To achieve the minimum discrepancy between theoretical data and observed data (inversion) theoretically requires N iterations raising the cost to N^3.

Consider an image of size $m \times m = N$. Continuing, for simplicity, to assume a dense matrix of relations between model and data, the cost for the adjoint is m^4 whereas the cost for inversion is m^6. We'll consider computational costs for the

year 2000, but noticing that costs go as the sixth power of the mesh size, the overall situation will not change much in the foreseeable future. Suppose you give a stiff workout to a powerful machine; you take an hour to invert a 4096×4096 matrix. The solution, a vector of 4096 components could be laid into an image of size $64 \times 64 = 2^6 \times 2^6 = 4096$. Here is what we are looking at for costs:

adjoint cost	$(m \times m)^2$	$(512 \times 512)^2$	$(2^9 2^9)^2$	2^{36}
inverse cost	$(m \times m)^3$	$(64 \times 64)^3$	$(2^6 2^6)^3$	2^{36}

These numbers tell us that for applications with dense operators, the biggest images that we are likely to see coming from inversion methods are 64×64 whereas those from adjoint methods are 512×512. For comparison, the retina of your eye is comparable to your computer screen at 1000×1000. We might summarize by saying that while adjoint methods are less than perfect, inverse methods are "legally blind" :-)

http://sepwww.stanford.edu/sep/jon/family/jos/gifmovie.html holds a movie blinking between Figures 3.1 and 3.2.

Figure 3.1: Jos greets Andrew, "Welcome back Andrew" from the Peace Corps. At a resolution of 512×512, this picture is about the same as the resolution as the paper it is printed on, or the same as your viewing screen, if you have scaled it to 50% of screen size. iin-512x512 [NR]

Figure 3.2: Jos greets Andrew, “Welcome back Andrew” again. At a resolution of 64×64 the pixels are clearly visible. From far the pictures are the same. From near, examine their glasses. iin-64x64 [NR]

This cost analysis is oversimplified in that most applications do not require dense operators. With sparse operators, the cost advantage of adjoints is even more pronounced since for adjoints, the cost savings of operator sparseness translate directly to real cost savings. The situation is less favorable and much more muddy for inversion. The reason that Chapter 2 covers iterative methods and neglects exact methods is that in practice iterative methods are not run to their theoretical completion but they run until we run out of patience.

Cost is a big part of the story, but the story has many other parts. Inversion, while being the only logical path to the best answer, is a path littered with pitfalls. The first pitfall is that the data is rarely able to determine a complete solution reliably. Generally there are aspects of the image that are not learnable from the data.

In this chapter we study the simplest, most transparant example of data insufficiency. Data exists at irregularly spaced positions in a plane. We set up a cartesian mesh and we discover that some of the bins contain no data points. What then?

3.1. MISSING DATA IN ONE DIMENSION

A method for restoring **missing data** is to ensure that the restored data, after specified filtering, has minimum energy. Specifying the filter chooses the interpolation philosophy. Generally the filter is a **roughening** filter. When a roughening filter goes off the end of smooth data, it typically produces a big end transient. Minimizing energy implies a choice for unknown data values at the end, to minimize the transient. We will examine five cases and then make some generalizations.

A method for restoring missing data is to ensure that the restored data, after specified filtering, has **minimum energy**.

Let u denote an unknown (missing) value. The dataset on which the examples are based is $(\cdots, u, u, 1, u, 2, 1, 2, u, u, \cdots)$. Theoretically we could adjust the missing u values (each different) to minimize the energy in the unfiltered data. Those adjusted values would obviously turn out to be all zeros. The unfiltered data is data that has been filtered by an impulse function. To find the missing values that minimize energy out of other filters, we can use subroutine `mis1()` /prog:mis1. Figure 3.3 shows interpolation of the dataset with $(1, -1)$ as a roughening filter.

The interpolated data matches the given data where they overlap.

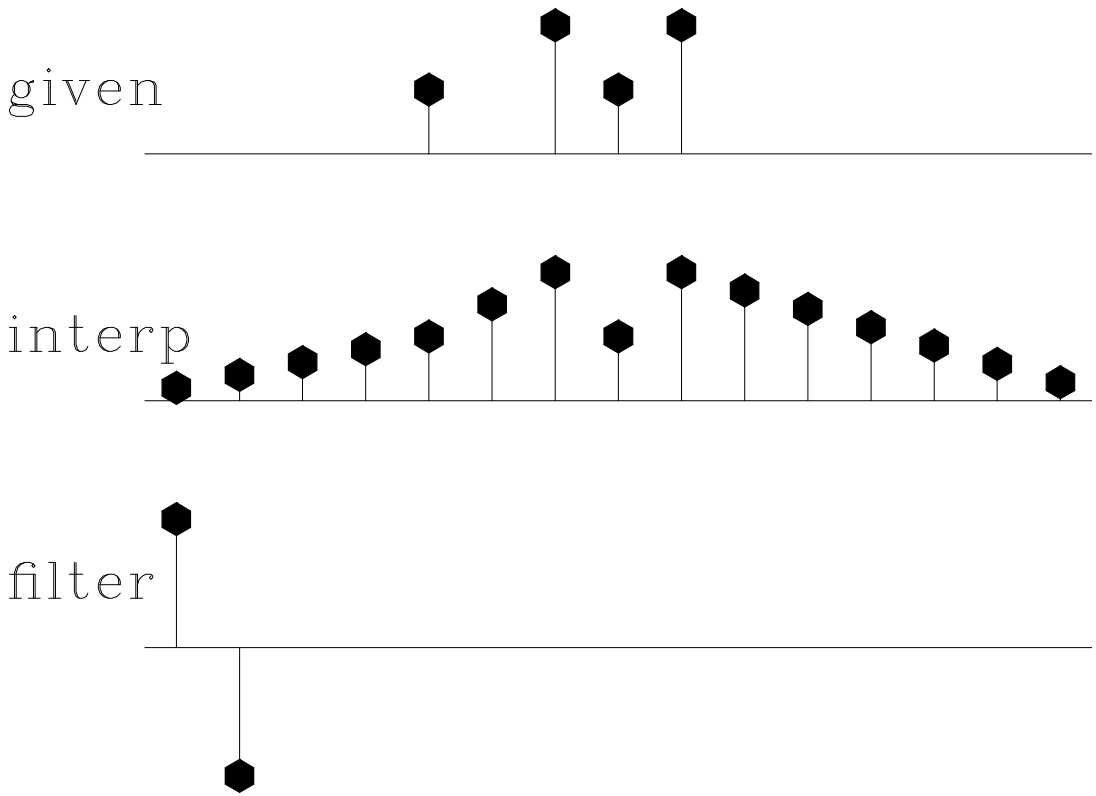


Figure 3.3: Top is given data. Middle is given data with interpolated values. Missing values seem to be interpolated by straight lines. Bottom shows the filter $(1, -1)$, whose output has minimum energy. iin-mlines90 [ER]

Figures 3.3–3.6 illustrate that the rougher the filter, the smoother the interpolated data, and vice versa. Let us switch our attention from the residual spectrum to the residual itself. The residual for Figure 3.3 is the *slope* of the signal (because the

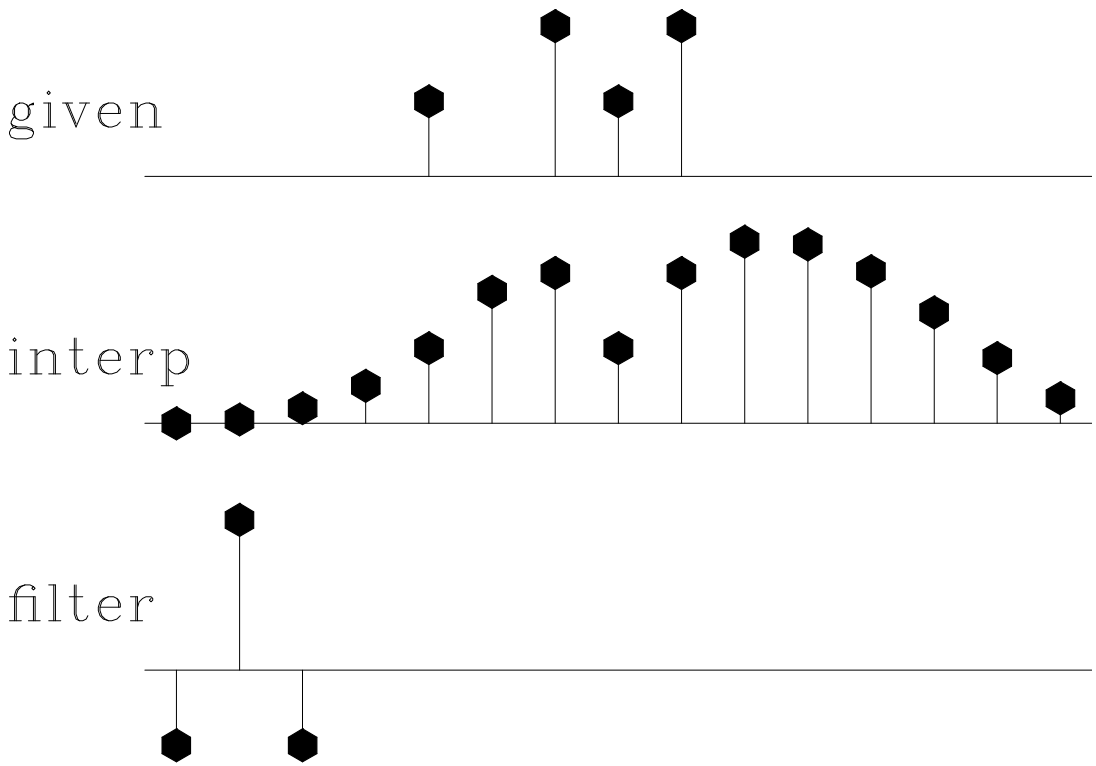


Figure 3.4: Top is the same input data as in Figure 3.3. Middle is interpolated. Bottom shows the filter $(-1, 2, -1)$. The missing data seems to be interpolated by parabolas. iin-mparab90 [ER]

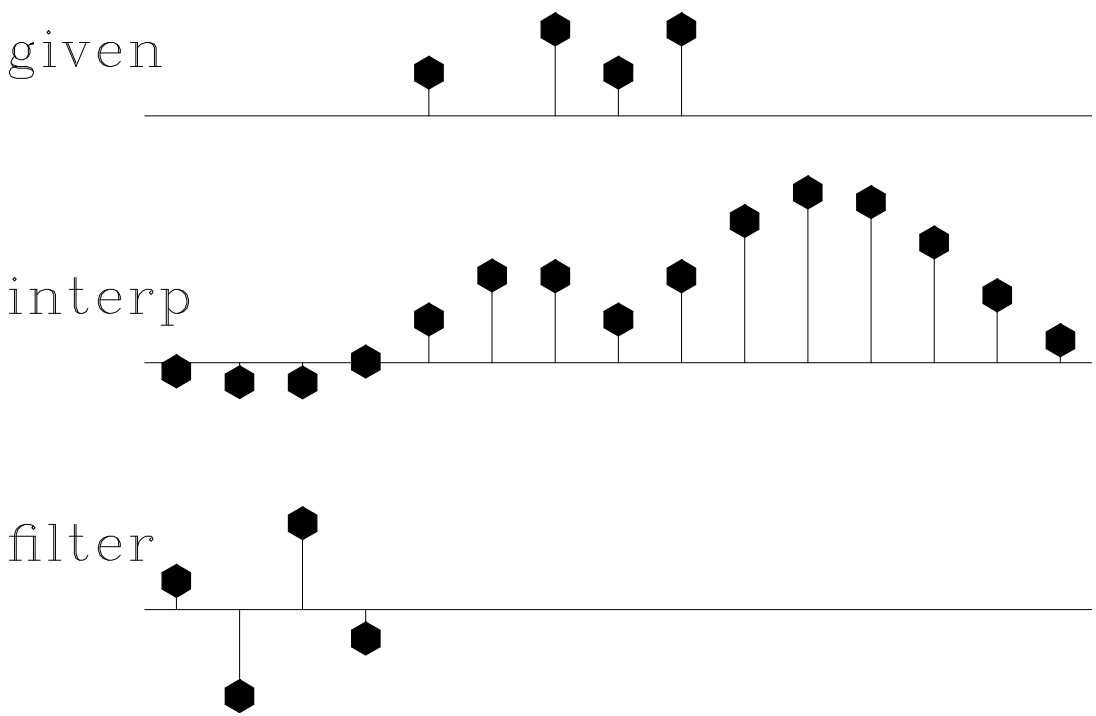


Figure 3.5: Top is the same input. Middle is interpolated. Bottom shows the filter $(1, -3, 3, -1)$. The missing data is very smooth. It shoots upward high off the right end of the observations, apparently to match the data slope there. iin-mseis90 [ER]

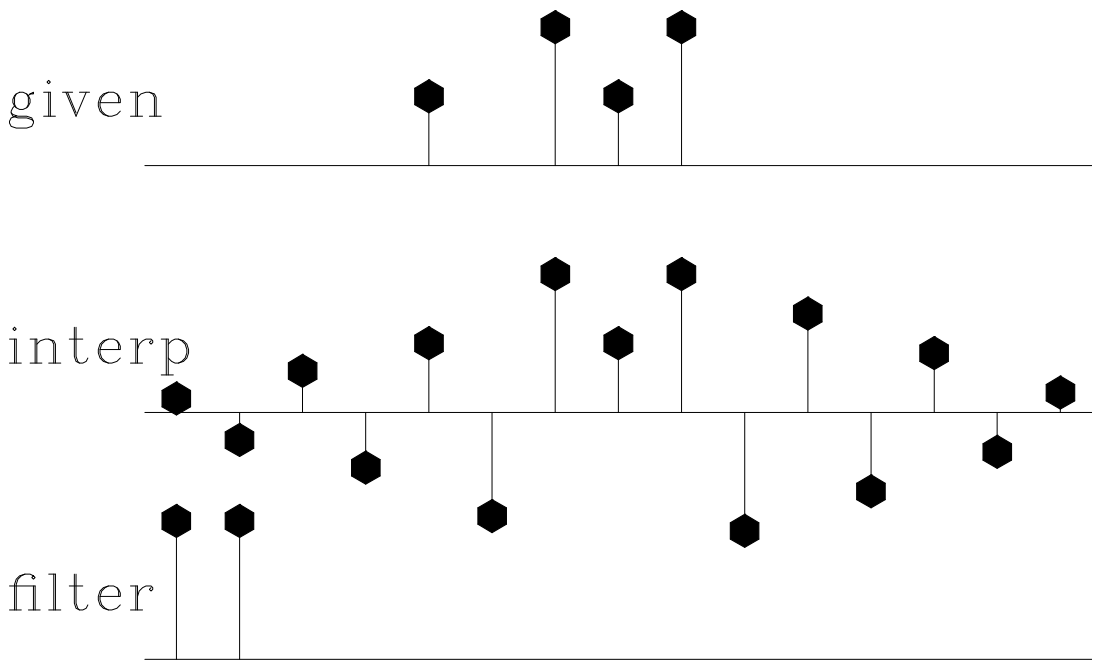


Figure 3.6: Bottom shows the filter $(1,1)$. The interpolation is rough. Like the given data itself, the interpolation has much energy at the Nyquist frequency. But unlike the given data, it has little zero-frequency energy. iin-moscil90 [ER]

filter $(1,-1)$ is a *first derivative*), and the slope is constant (uniformly distributed) along the straight lines where the least-squares procedure is choosing signal values. So these examples confirm the idea that the **least-squares method** abhors large values (because they are squared). Thus, least squares tends to distribute residuals uniformly in both time and frequency to the extent allowed by the **constraint**s.

This idea helps us answer the question, what is the best filter to use? It suggests choosing the filter to have an amplitude spectrum that is inverse to the spectrum we want for the interpolated data. A systematic approach is given in chapter 6, but I offer a simple subjective analysis here: Looking at the data, we see that all points are positive. It seems, therefore, that the data is rich in low frequencies; thus the filter should contain something like $(1,-1)$, which vanishes at zero frequency. Likewise, the data seems to contain Nyquist frequency, so the filter should contain $(1,1)$. The result of using the filter $(1,-1)*(1,1)=(1,0,-1)$ is shown in Figure 3.7. This is my best subjective interpolation based on the idea that the missing data should look like the given data. The **interpolation** and **extrapolation**s are so good that you can hardly guess which data values are given and which are interpolated.

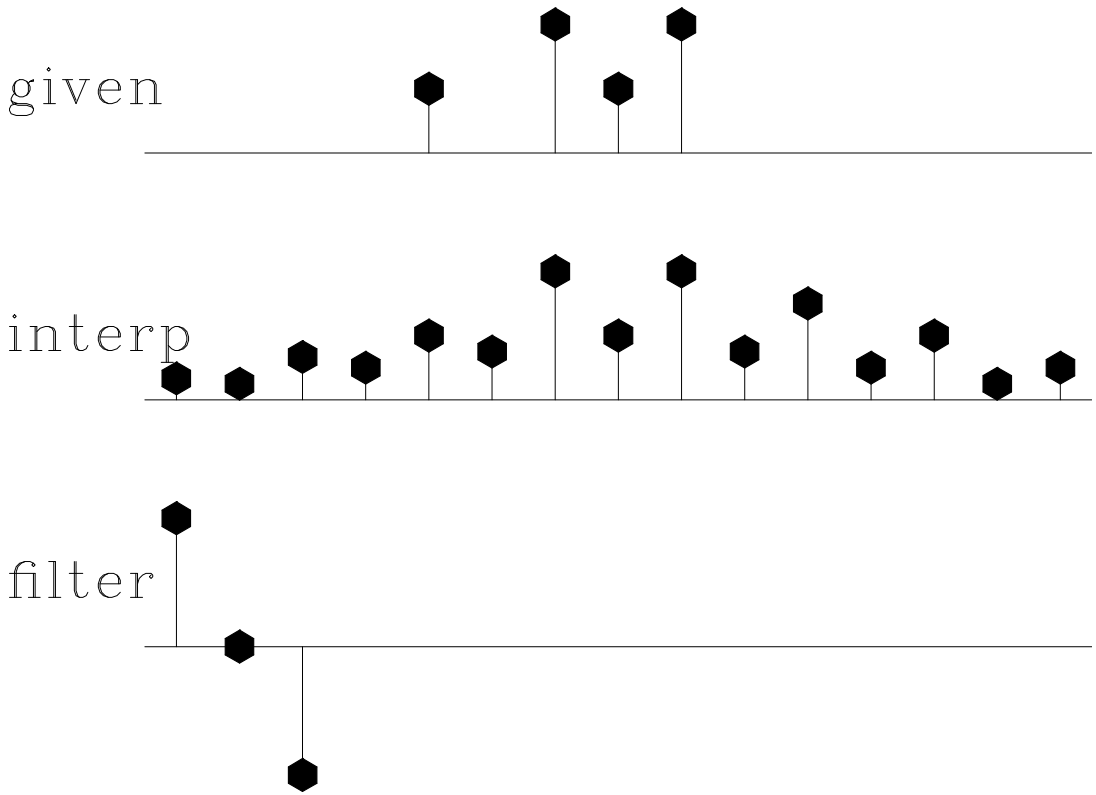


Figure 3.7: Top is the same as in Figures 3.3 to 3.6. Middle is interpolated. Bottom shows the filter $(1, 0, -1)$, which comes from the coefficients of $(1, -1) * (1, 1)$. Both the given data and the interpolated data have significant energy at both zero and Nyquist frequencies. iin-mbest90 [ER]

3.1.1. Missing-data program

Now let us see how Figures 3.3-3.7 could have been calculated and how they were calculated. They could have been calculated with matrices, in which matrices were pulled apart according to subscripts of known or missing data; instead I computed them with operators, and applied only operators and their adjoints. First we inspect the matrix approach because it is more conventional.

- **Matrix approach to missing data**

Customarily, we have referred to data by the symbol **d**. Now that we are dividing the data space into two parts, known and unknown (or missing), we will refer to this complete space as the model (or map) space **m**.

There are 15 data points in Figures 3.3-3.7. Of the 15, 4 are known and 11 are missing. Denote the known by k and the missing by u. Then the sequence of missing and known is $(u,u,u,u,k,u,k,k,k,u,u,u,u,u,u)$. Because I cannot print 15×15 matrices, please allow me to describe instead a data space of 6 values $(m_1,m_2,m_3,m_4,m_5,m_6)$ with known values only m_2 and m_3, that is arranged like (u,k,k,u,u,u).

Our approach is to minimize the energy in the residual, which is the filtered

map (model) space. We state the fitting goals $\mathbf{0} \approx \mathbf{Fm}$ as

$$\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \approx \mathbf{r} = \begin{bmatrix} a_1 & 0 & 0 & 0 & 0 & 0 \\ a_2 & a_1 & 0 & 0 & 0 & 0 \\ a_3 & a_2 & a_1 & 0 & 0 & 0 \\ 0 & a_3 & a_2 & a_1 & 0 & 0 \\ 0 & 0 & a_3 & a_2 & a_1 & 0 \\ 0 & 0 & 0 & a_3 & a_2 & a_1 \\ 0 & 0 & 0 & 0 & a_3 & a_2 \\ 0 & 0 & 0 & 0 & 0 & a_3 \end{bmatrix} \begin{bmatrix} m_1 \\ m_2 \\ m_3 \\ m_4 \\ m_5 \\ m_6 \end{bmatrix} \tag{3.1}$$

We rearrange the above fitting goals, bringing the columns multiplying known data

values (m_2 and m_3) to the left, getting $\mathbf{y} = -\mathbf{F}_k \mathbf{m}_k \approx \mathbf{F}_u \mathbf{m}_u$.

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \end{bmatrix} = - \begin{bmatrix} 0 & 0 \\ a_1 & 0 \\ a_2 & a_1 \\ a_3 & a_2 \\ 0 & a_3 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} m_2 \\ m_3 \end{bmatrix} \approx \begin{bmatrix} a_1 & 0 & 0 & 0 \\ a_2 & 0 & 0 & 0 \\ a_3 & 0 & 0 & 0 \\ 0 & a_1 & 0 & 0 \\ 0 & a_2 & a_1 & 0 \\ 0 & a_3 & a_2 & a_1 \\ 0 & 0 & a_3 & a_2 \\ 0 & 0 & 0 & a_3 \end{bmatrix} \begin{bmatrix} m_1 \\ m_4 \\ m_5 \\ m_6 \end{bmatrix} \tag{3.2}$$

This is the familiar form of an overdetermined system of equations $\mathbf{y} \approx \mathbf{F}_u \mathbf{m}_u$ which we could solve for $\mathbf{m}_u$ as illustrated earlier by conjugate directions, or by a wide variety of well-known methods.

The trouble with this matrix approach is that it is awkward to program the partitioning of the operator into the known and missing parts, particularly if the application of the operator uses arcane techniques, such as those used by the fast–Fourier-transform operator or various numerical approximations to differential or

partial differential operators that depend on regular data sampling. Even for the modest convolution operator, we already have a library of convolution programs that handle a variety of end effects, and it would be much nicer to use the library as it is rather than recode it for all possible geometrical arrangements of missing data values.

Note: Here I take the main goal to be the clarity of the code, not the efficiency or accuracy of the solution. So, if your problem consumes too many resources, and if you have many more known points than missing ones, maybe you should fit $\mathbf{y} \approx \mathbf{F}_u \mathbf{m}_u$ and ignore the suggestions below.

- **Operator approach to missing data**

For the operator approach to the fitting goal $-\mathbf{F}_k\mathbf{m}_k \approx \mathbf{F}_u\mathbf{m}_u$ we rewrite it as $-\mathbf{F}_k\mathbf{m}_k \approx \mathbf{FJm}$ where

$$-\mathbf{F}_k\mathbf{m}_k \quad \approx \quad \begin{bmatrix} a_1 & 0 & 0 & 0 & 0 & 0 \\ a_2 & a_1 & 0 & 0 & 0 & 0 \\ a_3 & a_2 & a_1 & 0 & 0 & 0 \\ 0 & a_3 & a_2 & a_1 & 0 & 0 \\ 0 & 0 & a_3 & a_2 & a_1 & 0 \\ 0 & 0 & 0 & a_3 & a_2 & a_1 \\ 0 & 0 & 0 & 0 & a_3 & a_2 \\ 0 & 0 & 0 & 0 & 0 & a_3 \end{bmatrix} \begin{bmatrix} 1 & . & . & . & . & . \\ . & 0 & . & . & . & . \\ . & . & 0 & . & . & . \\ . & . & . & 1 & . & . \\ . & . & . & . & 1 & . \\ . & . & . & . & . & 1 \end{bmatrix} \begin{bmatrix} m_1 \\ m_2 \\ m_3 \\ m_4 \\ m_5 \\ m_6 \end{bmatrix} \tag{3.3}$$

Notice the introduction of the new diagonal matrix **J**, called a **mask**ing matrix or a **constraint-mask** matrix because it multiplies constrained variables by zero leaving freely adjustable variables untouched. Experience shows that a better name than "mask matrix" is "**selector** matrix" because what comes out of it, that which is selected, is a less-confusing name for it than which is rejected. With a selector ma-

trix the whole data space seems freely adjustable, both the missing data values and known values. We see that the CD method does not change the known (constrained) values. In general, we derive the fitting goal (3.3) by

$$\mathbf{0} \approx \mathbf{F}\mathbf{m} \tag{3.4}$$

$$\mathbf{0} \approx \mathbf{F}(\mathbf{J}+(\mathbf{I}-\mathbf{J}))\mathbf{m} \tag{3.5}$$

$$\mathbf{0} \approx \mathbf{F}\mathbf{J}\mathbf{m}+\mathbf{F}(\mathbf{I}-\mathbf{J})\mathbf{m} \tag{3.6}$$

$$\mathbf{0} \approx \mathbf{F}\mathbf{J}\mathbf{m}+\mathbf{F}\mathbf{m}_{\text{known}} \tag{3.7}$$

$$\mathbf{0} \approx \mathbf{r} = \mathbf{F}\mathbf{J}\mathbf{m}+\mathbf{r}_0 \tag{3.8}$$

As usual, we find a direction to go $\Delta\mathbf{m}$ by the gradient of the residual energy.

$$\Delta\mathbf{m} = \frac{\partial}{\partial \mathbf{m}'}\,\mathbf{r}'\mathbf{r} = \left(\frac{\partial}{\partial \mathbf{m}'}\,\mathbf{r}'\right)\mathbf{r} = \left(\frac{\partial}{\partial \mathbf{m}'}\,(\mathbf{m}'\mathbf{J}'\mathbf{F}'+\mathbf{r}_0')\right)\mathbf{r} = \mathbf{J}'\mathbf{F}'\mathbf{r} \tag{3.9}$$

We begin the calculation with the known data values where missing data values are replaced by zeros, namely $(\mathbf{I}-\mathbf{J})\mathbf{m}$. Filter this data, getting $\mathbf{F}(\mathbf{I}-\mathbf{J})\mathbf{m}$, and load it into the residual $\mathbf{r}_0$. With this initialization completed, we begin an iteration loop.

First we compute Δm from equation (3.9).

$$\Delta \mathbf{m} \quad \longleftarrow \quad \mathbf{J}'\mathbf{F}'\mathbf{r} \tag{3.10}$$

$\mathbf{F}'$ applies a *crosscorrelation* of the filter to the residual and then $\mathbf{J}'$ sets to zero any changes proposed to known data values. Next, compute the change in residual $\Delta\mathbf{r}$ from the proposed change in the data $\Delta\mathbf{m}$.

$$\Delta \mathbf{r} \quad \longleftarrow \quad \mathbf{F}\mathbf{J}\Delta\mathbf{m} \tag{3.11}$$

This applies the filtering again. Then use the method of steepest descent (or conjugate direction) to choose the appropriate scaling (or inclusion of previous step) of $\Delta\mathbf{m}$ and $\Delta\mathbf{r}$, and update $\mathbf{m}$ and $\mathbf{r}$ accordingly and iterate.

I could have passed a new operator $\mathbf{FJ}$ into the old solver, but found it worthwhile to write a new, more powerful solver having built-in constraints. To introduce the masking operator $\mathbf{J}$ into the `solver_smp` subroutine /prog:solver_tiny, I introduce an optional operator `Jop`, which is initialized with a logical array of the model size. Two lines in the `solver_tiny` module /prog:solver_tiny

```
stat = Fop( T, F, g, rd)                    #  g = F' Rd
stat = Fop( F, F, g, gd)                    #  G = F  g
```

become three lines in the standard library module `solver_smp`. (We use a temporary array `tm` of the size of model space.) $\Delta\mathbf{m}$ is `g` and $\Delta\mathbf{r}$ is `gg`.

```
stat = Fop( T, F, g, rd)                                  # g = F' Rd
if ( present( Jop)) { tm=g;  stat= Jop( F, F, tm, g)      # g = J g
stat = Fop( F, F, g, gg)                                  # G = F g
```

The full code includes all the definitions we had earlier in `solver_tiny` module /prog:solver_tiny. Merging it with the above bits of code we have the simple solver `solver_smp`. solver_smp

There are two methods of invoking the solver. Comment cards in the code indicate the slightly more verbose method of solution which matches the theory presented in the book.

The subroutine to find missing data is `mis1()`. It assumes that zero values in the input data correspond to missing data locations. It uses our convolution operator `tcai1()` /prog:tcai1. You can also check the Index for other **operators** and **modules**. mis1

I sought reference material on conjugate gradients with constraints and didn't find anything, leaving me to fear that this chapter was in error and that I had lost

```
module solver_smp_mod {                          # 0 = W (F J m - d)
  use chain0_mod + solver_report_mod
  logical, parameter, private  :: T = .true., F = .false.
contains
  subroutine solver_smp( m,d, Fop, stepper, niter &
  ,             Wop,Jop,m0,err,resd,mmov,rmov,verb) {
    optional :: Wop,Jop,m0,err,resd,mmov,rmov,verb
    interface { #-------------------------- begin definitions -----------
       integer function Fop(adj,add,m,d){real::m(:),d(:);logical,intent(in)::adj,add}
       integer function Wop(adj,add,m,d){real::m(:),d(:);logical,intent(in)::adj,add}
       integer function Jop(adj,add,m,d){real::m(:),d(:);logical,intent(in)::adj,add}
       integer function stepper(forget,m,g,rr,gg) {
         real, dimension(:)  ::        m,g,rr,gg
         logical             :: forget           }
    }
    real, dimension(:),    intent(in)          :: d, m0
    integer,               intent(in)          :: niter
    logical,               intent(in)          :: verb
    real, dimension(:),    intent(out)         :: m,err, resd
    real, dimension(:,:),  intent(out)         ::        rmov, mmov
    real, dimension(size(m))                   :: g
    real, dimension(size(d)),         target   :: rr, gg
    real, dimension(size(d)+size(m)), target   :: tt
    real, dimension(:), pointer                :: rd, gd, td
    real, dimension(:), pointer                :: rm, gm, tm
    integer                                    :: iter, stat
    logical                                    :: forget
    rd => rr(1:size(d));
    gd => gg(1:size(d));
    td => tt(1:size(d)); tm => tt(1+size(d):)
    if(present( Wop)) stat=Wop(F,F,-d,rd) # begin initialization --------
    else rd = -d                                          #Rd = -W  d
    if(present( m0)){ m=m0                                #m  =     m0
        if(present( Wop)) call chain0(Wop,Fop,F,T,m,rd,td)
        else              stat =           Fop(F,T,m,rd   ) #Rd+=  WF m0
    } else m=0
    forget = T; #-------------------------- begin iterations -----------
    do iter = 1,niter {
```

```
module mis_mod {
  use tcai1+mask1+cgstep_mod+solver_smp_mod
#  use mtcai1
contains
# fill in missing data on 1-axis by minimizing power out of a given filter.
  subroutine mis1 ( niter, mm, aa) {
    integer,              intent (in)      :: niter   # number of iterations
    real, dimension (:), pointer           :: aa      # roughening filter
    real, dimension (:), intent (in out)   :: mm      # in - data with zeroes
                                                      # out - interpolated
    real, dimension (:),allocatable        :: zero    # filter output
    logical, dimension(:), pointer         :: msk
    integer                                :: stat
#    real, dimension (:),allocatable        :: dd
    allocate(zero(size(mm)+size(aa))); zero = 0.
    allocate( msk(size(mm)))
#    allocate(  dd(size(mm)+size(aa)))
    # solve    F    m = 0  w/ J
    msk=(mm==0.); call mask1_init(msk)
    call tcai1_init(aa)
    call solver_smp(m=mm,d=zero,Fop=tcai1_lop,stepper=cgstep,niter=niter,m0=mm,Jop=mask
    # solve  (F J) m = d
#    call mtcai1_init(aa,msk)                  #       F(I-J)
#    stat = mtcai1_lop(.false.,.false.,mm,dd) #       F(I-J) m
#    dd = - dd                                 # d = - F(I-J) m
#    msk=(mm==0.); call mask1_init(msk)        #           J
#    call solver_smp(m=mm,d=dd,Fop=mtcai1_lop,stepper=cgstep,niter=niter,m0=mm)
    call cgstep_close ()
    deallocate(zero)
    }
}
```

Back

the magic property of convergence in a finite number of iterations. I tested the code and it did converge in a finite number of iterations. The explanation is that these constraints are almost trivial. We pretended we had extra variables, and computed a $\Delta\mathbf{m} = \mathbf{g}$ for each of them. Then we set the $\Delta\mathbf{m} = \mathbf{g}$ to zero, hence making no changes to anything, like as if we had never calculated the extra $\Delta\mathbf{m}$'s.

EXERCISES:

1 Figures 3.3–3.6 seem to extrapolate to vanishing signals at the side boundaries. Why is that so, and what could be done to leave the sides unconstrained in that way?

2 Show that the interpolation curve in Figure 3.4 is not parabolic as it appears, but cubic. (HINT: First show that $(\nabla^2)'\nabla^2 u = \mathbf{0}$.)

3 Verify by a program example that the number of iterations required with simple constraints is the number of free parameters.

4 A signal on a uniform mesh has missing values. How should we estimate the mean?

3.2. WELLS NOT MATCHING THE SEISMIC MAP

Accurate knowledge comes from a **well**, but wells are expensive and far apart. Less accurate knowledge comes from surface seismology, but this knowledge is available densely in space and can indicate significant **trend**s between the wells. For example, a prospective area may contain 15 wells but 600 or more seismic stations. To choose future well locations, it is helpful to match the known well data with the seismic data. Although the seismic data is delightfully dense in space, it often mismatches the wells because there are systematic differences in the nature of the measurements. These discrepancies are sometimes attributed to velocity **anisotropy**. To work with such measurements, we do not need to track down the physical model, we need only to merge the information somehow so we can appropriately **map** the trends between wells and make a proposal for the next drill site. Here we consider only a scalar value at each location. Take **w** to be a vector of 15 components, each component being the seismic travel time to some fixed depth in a well. Likewise let **s** be a 600-component vector each with the seismic travel time to that fixed depth as estimated wholly from surface seismology. Such empirical corrections are often

called "**fudge factor**s". An example is the Chevron oil field in Figure 3.8. The binning of the seismic data in Figure 3.8 is not really satisfactory when we have available the techniques of missing data estimation to fill the empty bins. Using the ideas of subroutine `mis1()` /prog:mis1, we can extend the seismic data into the empty part of the plane. We use the same principle that we minimize the energy in the filtered map where the map must match the data where it is known. I chose the filter $\mathbf{A} = \nabla' \nabla = -\nabla^2$ to be the Laplacian operator (actually, its negative) to obtain the result in Figure 3.9.

Figure 3.9 also involves a **boundary condition calculation**. Many differential equations have a solution that becomes infinite at infinite distance, and in practice this means that the largest solutions may often be found on the boundaries of the plot, exactly where there is the least information. To obtain a more pleasing result, I placed artificial "average" data along the outer boundary. Each boundary point was given the value of an average of the interior data values. The average was weighted, each weight being an inverse power of the separation distance of the boundary point from the interior point.

Parenthetically, we notice that all the unknown interior points could be guessed by the same method we used on the outer boundary. After some experience guessing

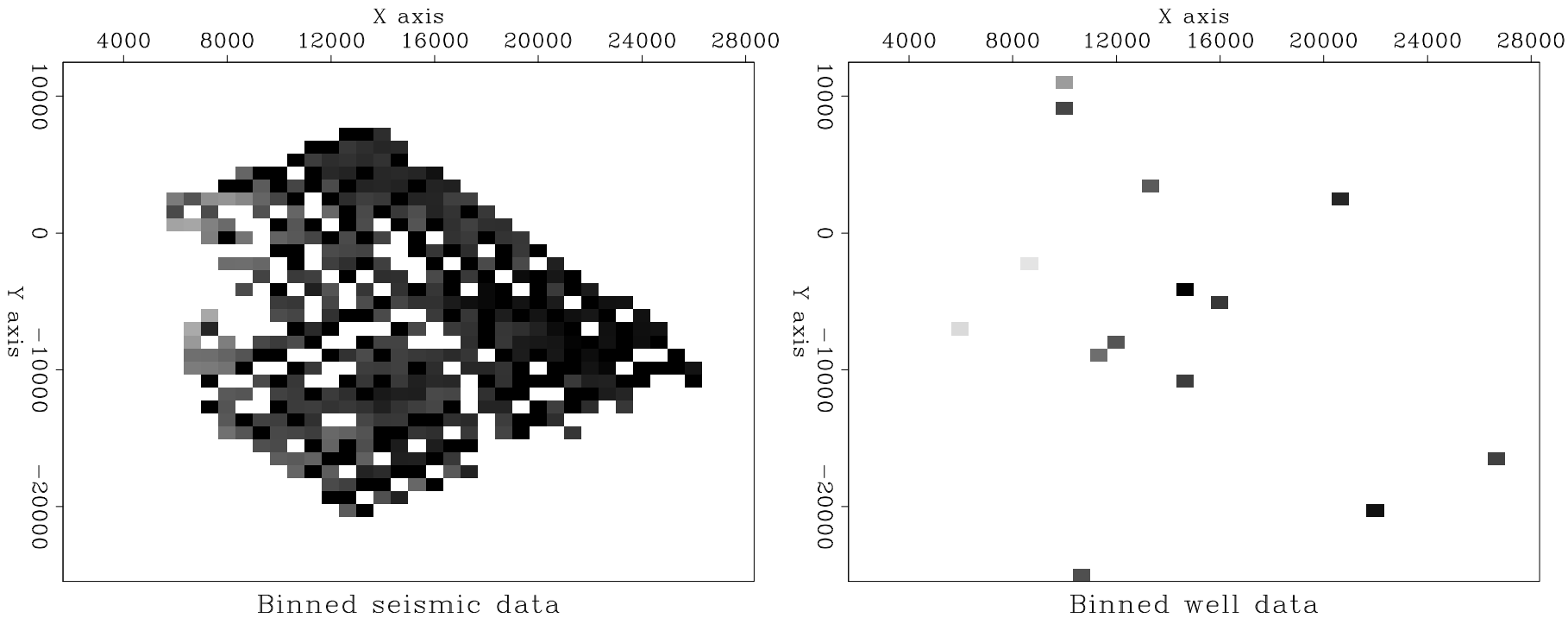


Figure 3.8: Binning by data push. Left is seismic data. Right is well locations. Values in bins are divided by numbers in bins. (Toldi) iin-wellseis90 [ER]

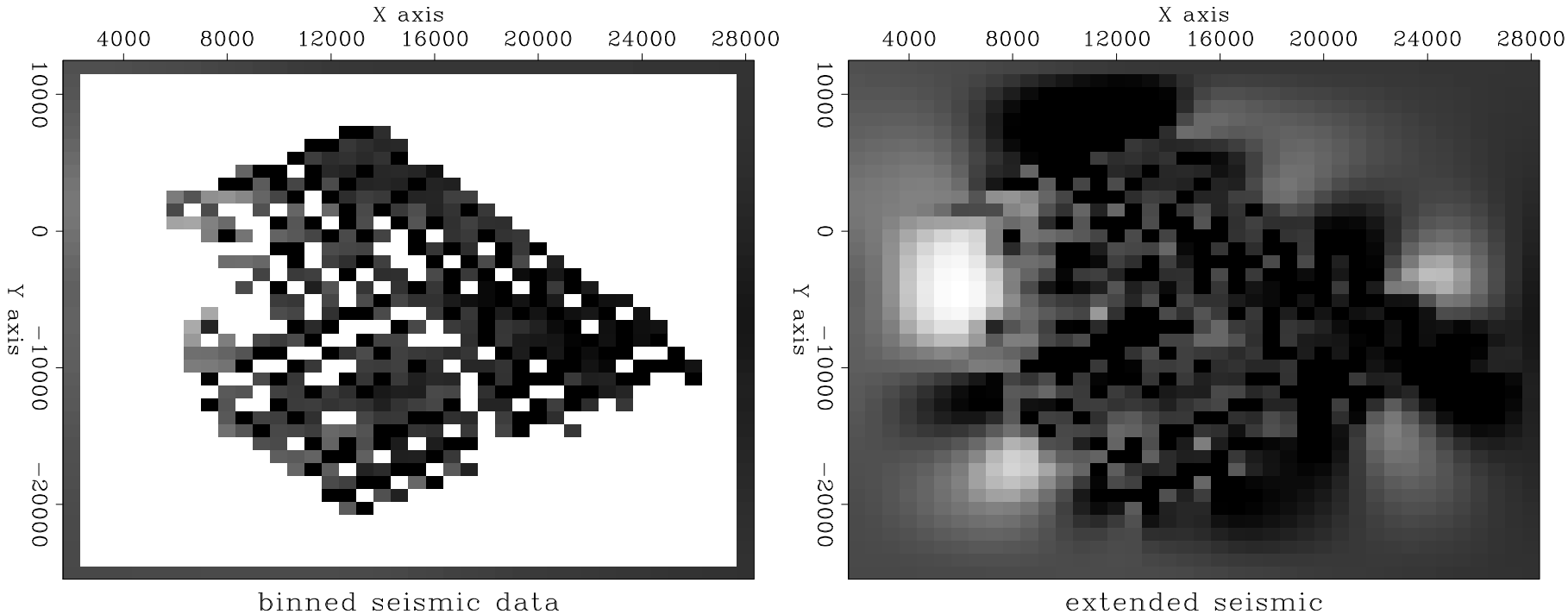


Figure 3.9: Seismic binned (left) and extended (right) by minimizing energy in $\nabla^2\mathbf{s}$. iin-misseis90 [ER]

what inverse power would be best for the weighting functions, I do not recommend this method. Like gravity, the forces of interpolation from the weighted sums are not blocked by intervening objects. But the temperature in a house is not a function of temperature in its neighbor's house. To further isolate the more remote points, I chose weights to be the inverse fourth power of distance.

The first job is to fill the gaps in the seismic data. We just finished doing a job like this in one dimension. I'll give you more computational details later. Let us call the extended seismic data **s**.

Think of a map of a model space **m** of infinitely many hypothetical wells that must match the real wells, where we have real wells. We must find a map that matches the wells exactly and somehow matches the seismic information elsewhere. Let us define the vector **w** as shown in Figure 3.8 so **w** is observed values at wells and zeros elsewhere.

Where the seismic data contains sharp bumps or streaks, we want our final earth model to have those features. The wells cannot provide the rough features because the wells are too far apart to provide high spatial frequencies. The well information generally conflicts with the seismic data at low spatial frequencies because of systematic discrepancies between the two types of measurements. Thus we must accept

that **m** and **s** may differ at low spatial frequencies (where gradient and Laplacian are small).

Our final map **m** would be very unconvincing if it simply jumped from a well value at one point to a seismic value at a neighboring point. The map would contain discontinuities around each well. Our philosophy of finding an earth model **m** is that our earth map should contain no obvious "footprint" of the data acquisition (well locations). We adopt the philosopy that the difference between the final map (extended wells) and the seismic information $\mathbf{x} = \mathbf{m} - \mathbf{s}$ should be smooth. Thus, we seek the minimum residual **r** which is the roughened difference between the seismic data **s** and the map **m** of hypothetical omnipresent wells. With roughening operator **A** we fit

$$\mathbf{0} \quad \approx \quad \mathbf{r} \quad = \quad \mathbf{A}(\mathbf{m}-\mathbf{s}) \quad = \quad \mathbf{Ax} \tag{3.12}$$

along with the constraint that the map should match the wells at the wells. We could write this as $\mathbf{0} = (\mathbf{I} - \mathbf{J})(\mathbf{m} - \mathbf{w})$. We honor this constraint by initializing the map $\mathbf{m} = \mathbf{w}$ to the wells (where we have wells, and zero elsewhere). After we find the gradient direction to suggest some changes to **m**, we simply will not allow those changes at well locations. We do this with a mask. We apply a "missing data

selector" to the gradient. It zeros out possible changes at well locations. Like with the goal (3.7), we have

$$\mathbf{0} \quad \approx \quad \mathbf{r} \quad = \quad \mathbf{AJx} + \mathbf{Ax}_{\text{known}} \tag{3.13}$$

After minimizing $\mathbf{r}$ by adjusting $\mathbf{x}$, we have our solution $\mathbf{m} = \mathbf{x} + \mathbf{s}$.

Now we prepare some roughening operators $\mathbf{A}$. We have already coded a 2-D gradient operator `igrad2` /prog:igrad2. Let us combine it with its adjoint to get the 2-D laplacian operator. (You might notice that the laplacian operator is "self-adjoint" meaning that the operator does the same calculation that its adjoint does. Any operator of the form $\mathbf{A}'\mathbf{A}$ is self-adjoint because $(\mathbf{A}'\mathbf{A})' = \mathbf{A}'\mathbf{A}'' = \mathbf{A}'\mathbf{A}$.)

laplac2 Subroutine `lapfill2()` /prog:lapfill2 is the same idea as `mis1()` /prog:mis1 except that the filter $\mathbf{A}$ has been specialized to the laplacian implemented by module `laplac2` /prog:laplac2. lapfill2

Subroutine `lapfill2()` can be used for each of our two problems, (1) extending the seismic data to fill space, and (2) fitting the map exactly to the wells and approximately to the seismic data. When extending the seismic data, the initially non-zero components $\mathbf{s} \neq \mathbf{0}$ are fixed and cannot be changed. That is done by calling `lapfill2()` with `mfixed=(s/=0.)`. When extending wells, the initially non-

```
module laplac2 {                                        # Laplacian operator in 2-D
use igrad2
logical, parameter, private  :: T = .true., F = .false.
real, dimension (m1*m2*2), allocatable :: tmp
#%_init    (m1, m2)
    integer m1, m2
    call igrad2_init (m1, m2)
#%_lop (x, y)
    integer stat1, stat2
    if( adj) {
        stat1 = igrad2_lop (  F,    F, y, tmp)   # tmp = grad y
        stat2 = igrad2_lop (  T,  add, x, tmp)   # x = x + grad' tmp
    } else   {
        stat1 = igrad2_lop (  F,    F, x, tmp)   # tmp = grad x
        stat2 = igrad2_lop (  T,  add, y, tmp)   # y = y + grad' tmp
    }
}
```

Back

```
module lapfill {  # fill empty 2-D bins by minimum output of Laplacian operator
  use laplac2
  use cgstep_mod
  use mask1
  use solver_smp_mod
contains
  subroutine lapfill2( niter, m1, m2, yy, mfixed) {
    integer,                    intent (in)     :: niter, m1, m2
    logical, dimension (m1*m2), intent (in)     :: mfixed # mask for known
    real,    dimension (m1*m2), intent (in out) :: yy     # model
    real,    dimension (m1*m2)                  :: zero   # laplacian output
    logical, dimension (:),     pointer         :: msk
    allocate(msk(size(mfixed)))
    msk=.not.mfixed
    call mask1_init(msk)
    call laplac2_init ( m1,m2);             zero = 0.      # initialize
    call solver_smp(m=yy, d=zero, Fop=laplac2_lop, step-
per=cgstep, niter=niter, m0=yy, Jop=mask1_lop)
    call laplac2_close ()                                  # garbage collection
    call cgstep_close ()                                   # garbage collection
  }
}
```

Back

zero components $\mathbf{w} \neq \mathbf{0}$ are fixed and cannot be changed. That is done by calling `lapfill2()` with `mfixed=(w/=0.)`.

The final map is shown in Figure 3.10.

Results can be computed with various filters. I tried both ∇^2 and ∇. There are disadvantages of each, ∇ being too cautious and ∇^2 perhaps being too aggressive. Figure 3.11 shows the difference $\mathbf{x}$ between the extended seismic data and the extended wells. Notice that for ∇ the difference shows a localized "tent pole" disturbance about each well. For ∇^2 there could be large overshoot between wells, especially if two nearby wells have significantly different values. I don't see that problem here.

My overall opinion is that the Laplacian does the better job in this case. I have that opinion because in viewing the extended gradient I can clearly see where the wells are. The wells are where we have acquired data. We'd like our map of the world to not show where we acquired data. Perhaps our estimated map of the world cannot help but show where we have and have not acquired data, but we'd like to minimize that aspect.

A good image of the earth hides our data **acquisition footprint**.

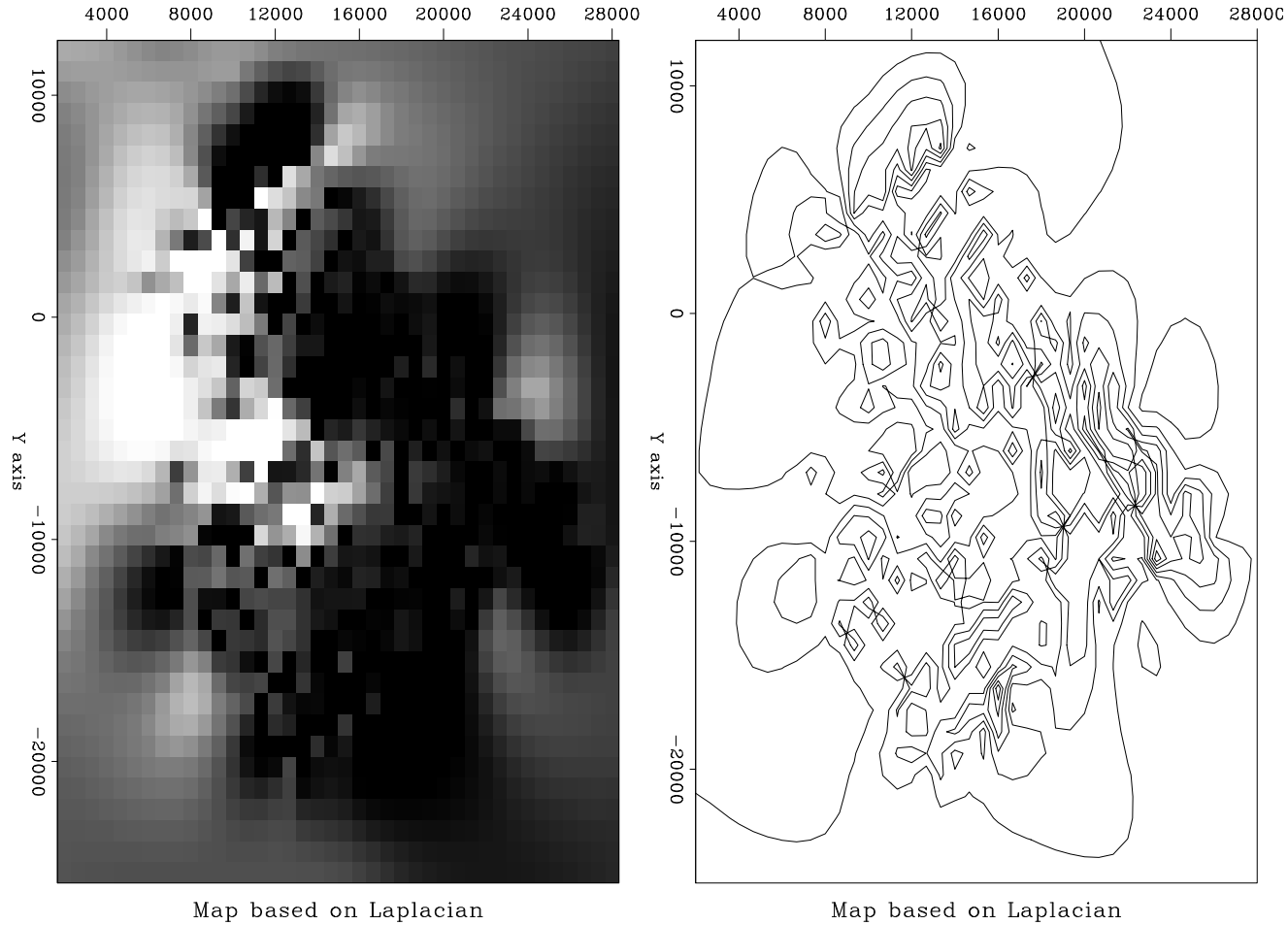


Figure 3.10: Final map based on Laplacian roughening. iin-finalmap90 [ER,M]

To understand the behavior theoretically, recall that in one dimension the filter ∇ interpolates with straight lines and ∇^2 interpolates with cubics. This is because the fitting goal $\mathbf{0} \approx \nabla\mathbf{m}$, leads to $\frac{\partial}{\partial \mathbf{m}'}\mathbf{m}'\nabla'\nabla\mathbf{m} = \mathbf{0}$ or $\nabla'\nabla\mathbf{m} = \mathbf{0}$, whereas the fitting goal $\mathbf{0} \approx \nabla^2\mathbf{m}$ leads to $\nabla^4\mathbf{m} = \mathbf{0}$ which is satisfied by cubics. In two dimensions, minimizing the output of ∇ gives us solutions of Laplace's equation with sources at the known data. It is as if ∇ stretches a rubber sheet over poles at each well, whereas ∇^2 bends a stiff plate.

Just because ∇^2 gives smoother maps than ∇ does not mean those maps are closer to reality. This is a deeper topic, addressed in Chapter 6. It is the same issue we noticed when comparing figures 3.3-3.7.

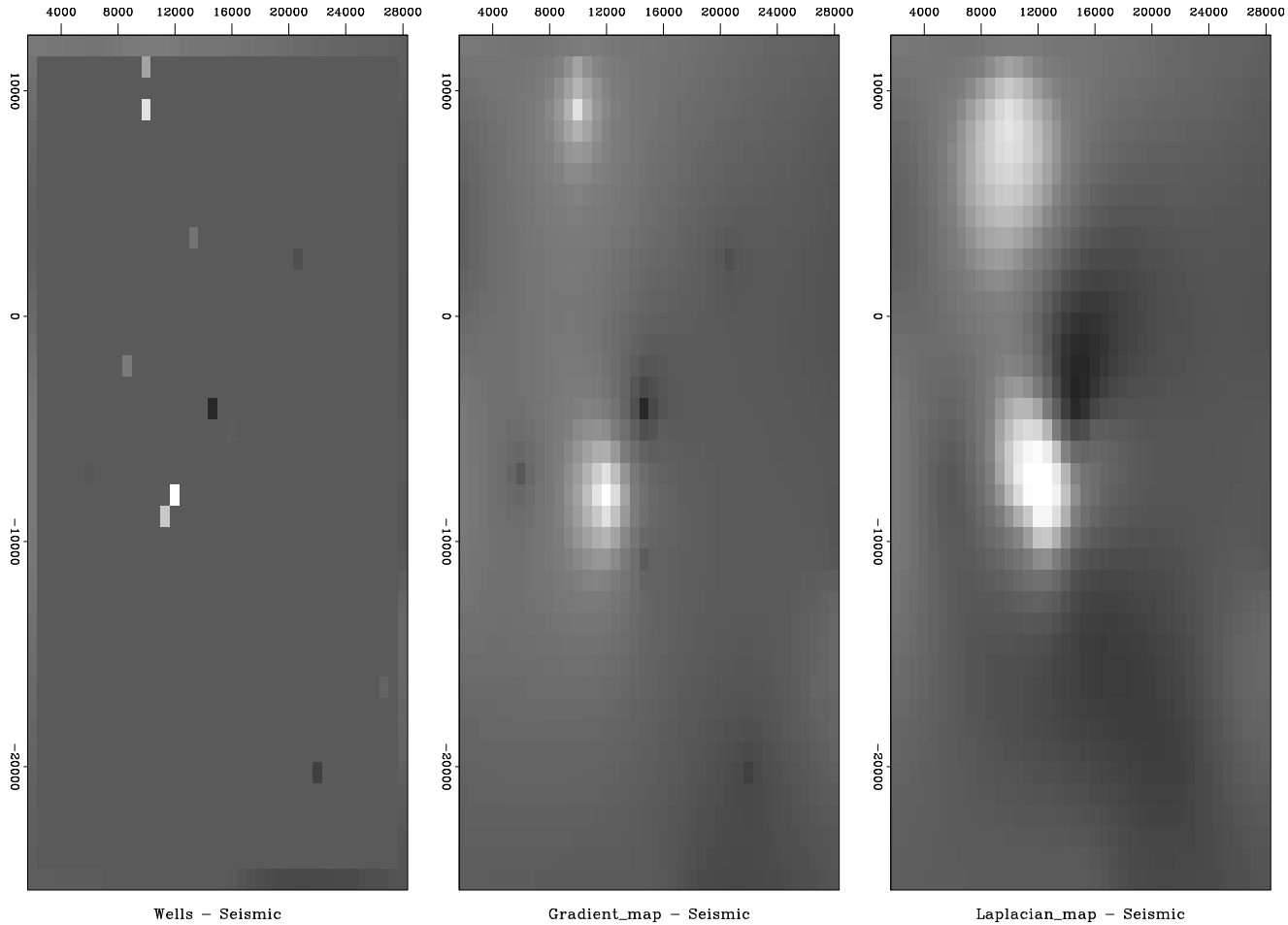


Figure 3.11: Difference between wells (the final map) and the extended seismic data. Left is plotted at the wells (with gray background for zero). Center is based on gradient roughening and shows tent-pole-like residuals at wells. Right is based on Laplacian roughening. iin-diffdiff90 [ER]

3.3. SEARCHING THE SEA OF GALILEE

Figure 3.12 shows a bottom-sounding survey of the Sea of Galilee[1] at various stages of processing. The ultimate goal is not only a good map of the depth to bottom, but images useful for the purpose of identifying **archaeological**, geological, or geophysical details of the sea bottom. The Sea of Galilee is unique because it is a *fresh*-water lake *below* sea-level. It seems to be connected to the great rift (pull-apart) valley crossing east Africa. We might delineate the Jordan River delta. We might find springs on the water bottom. We might find archaeological objects.

The raw data is 132,044 triples, (x_i, y_i, z_i), where x_i ranges over about 12 km and where y_i ranges over about 20 km. The lines you see in Figure 3.12 are sequences of data points, i.e., the track of the survey vessel. The depths z_i are recorded to an accuracy of about 10 cm.

The first frame in Figure 3.12 shows simple binning. A coarser mesh would avoid the empty bins but lose resolution. As we refine the mesh for more detail,

---

[1] Data collected by Zvi **ben Avraham**, TelAviv University. Please communicate with him `zvi@jupiter1.tau.ac.il` for more details or if you make something publishable with his data.

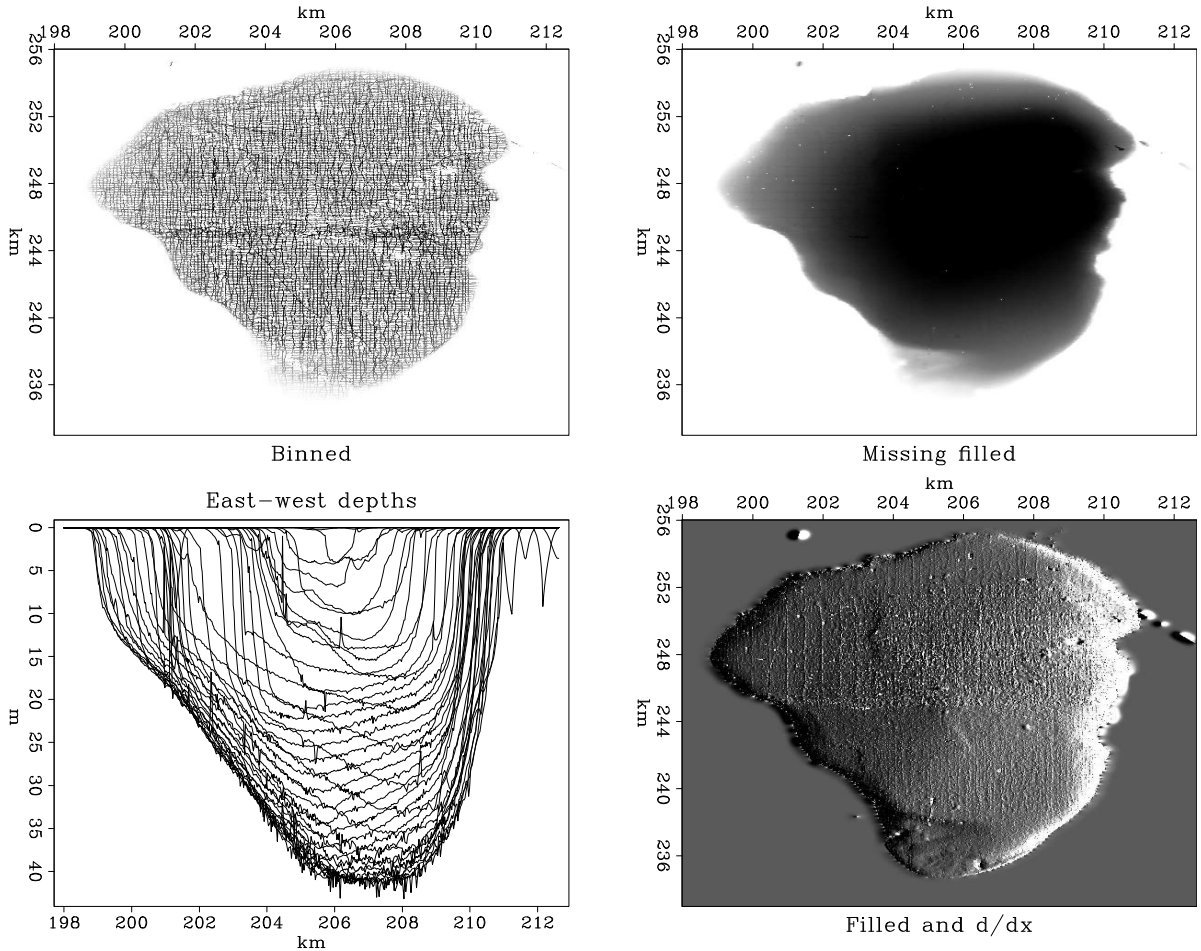


Figure 3.12: Views of the bottom of the Sea of Galilee. iin-locfil90 [ER,M]

```
module grad2fill {   # min r(m) = L J m + L known  where L is a lowcut filter.
  use igrad2
  use cgstep_mod
  use mask1
  use solver_smp_mod
contains
  subroutine grad2fill2( niter, m1, m2, mm, mfixed) {
    integer,                    intent (in)     :: niter, m1,m2
    logical, dimension (m1*m2), intent (in)     :: mfixed      # mask for known
    real,    dimension (m1*m2), intent (in out) :: mm          # model
    real,    dimension (m1*m2*2)                :: yy          # lowcut output
    logical, dimension (:),     pointer         :: msk
    allocate(msk(size(mfixed)))
    msk=.not.mfixed
    call mask1_init(msk)
    call igrad2_init(m1,m2);                       yy = 0.     # initialize
    call solver_smp(m=mm, d=yy, Fop=igrad2_lop, step-
per=cgstep, niter=niter, m0=mm, Jop=mask1_lop)
    call cgstep_close ()
  }
}
```

Back

the number of empty bins grows as does the care needed in devising a technique for filling them. This first frame uses the simple idea from Chapter 1 of spraying all the data values to the nearest bin with `bin2()` /prog:bin2 and dividing by the number in the bin. Bins with no data obviously need to be filled in some other way. I used a missing data program like that in the recent section on "wells not matching the seismic map." Instead of roughening with a Laplacian, however, I used the gradient operator `igrad2` /prog:igrad2 The solver is `grad2fill()`. grad2fill

The output of the roughening operator is an image, a filtered version of the depth, a filtered version of something real. Such filtering can enhance the appearance of interesting features. For example, scanning the shoreline of the roughened image (after missing data was filled), we see several ancient shorelines, now submerged.

> The adjoint is the easiest image to build. The roughened map is often more informative than the map itself.

The views expose several defects of the data acquisition and of our data processing. The impulsive glitches (St. Peter's fish?) need to be removed but we must be careful not to throw out the sunken ships along with the bad data points. Even

our best image shows clear evidence of the recording vessel's tracks. Strangely, some tracks are deeper than others. Perhaps the survey is assembled from work done in different seasons and the water level varied by season. Perhaps some days the vessel was more heavily loaded and the depth sounder was on a deeper keel. As for the navigation equipment, we can see that some data values are reported outside the lake!

We want the sharpest possible view of this classical site. A treasure hunt is never easy and no one guarantees we will find anything of great value but at least the exercise is a good warm-up for submarine petroleum exploration.

3.4. INVERSE LINEAR INTERPOLATION

In Chapter 1 we defined **linear interpolation** as the extraction of values from between mesh points. In a typical setup (occasionally the role of data and model are swapped), a model is given on a uniform mesh and we solve the easy problem of extracting values between the mesh points with subroutine `lint1()` /prog:lint1. The genuine problem is the inverse problem, which we attack here. Data values are sprinkled all around, and we wish to find a function on a uniform mesh from which

we can extract that data by **linear interpolation**. The adjoint operator for subroutine `lint1()` simply piles data back into its proper location in model space without regard to how many data values land in each region. Thus some model values may have many data points added to them while other model values get none. We could interpolate by minimizing the energy in the model gradient, or that in the second derivative of the model, or that in the output of any other roughening filter applied to the model.

Formalizing now our wish that data **d** be extractable by **linear interpolation F**, from a model **m**, and our wish that application of a roughening filter with an operator **A** have minimum energy, we write the fitting goals:

$$\begin{aligned} \mathbf{0} &\approx \mathbf{Fm} - \mathbf{d} \\ \mathbf{0} &\approx \mathbf{Am} \end{aligned} \tag{3.14}$$

Suppose we take the roughening filter to be the second difference operator $(1, -2, 1)$ scaled by a constant ϵ, and suppose we have a data point near each end of the model and a third data point exactly in the middle. Then, for a model space 6 points long,

the fitting goal could look like

$$\left[\begin{array}{cccccc} .8 & .2 & . & . & . & . \\ . & . & 1 & . & . & . \\ . & . & . & . & .5 & .5 \\ \hline \epsilon & . & . & . & . & . \\ -2\epsilon & \epsilon & . & . & . & . \\ \epsilon & -2\epsilon & \epsilon & . & . & . \\ . & \epsilon & -2\epsilon & \epsilon & . & . \\ . & . & \epsilon & -2\epsilon & \epsilon & . \\ . & . & . & \epsilon & -2\epsilon & \epsilon \\ . & . & . & . & \epsilon & -2\epsilon \\ . & . & . & . & . & \epsilon \end{array}\right] \left[\begin{array}{c} m_0 \\ m_1 \\ m_2 \\ m_3 \\ m_4 \\ m_5 \end{array}\right] - \left[\begin{array}{c} d_0 \\ d_1 \\ d_2 \\ \hline 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{array}\right] = \left[\begin{array}{c} \mathbf{r}_d \\ \mathbf{r}_m \end{array}\right] \approx \tag{3.15}$$

The residual vector has two parts, a data part $\mathbf{r}_d$ on top and a model part $\mathbf{r}_m$ on the bottom. The data residual should vanish except where contradictory data values happen to lie in the same place. The model residual is the roughened model.

Two fitting goals (3.14) are so common in practice that it is convenient to adopt

our least-square fitting subroutine `solver_smp` /prog:solver_smp accordingly. The modification is shown in module `solver_reg` /prog:solver_reg. In addition to specifying the "data fitting" operator **F** (parameter `Fop`), we need to pass the "model regularization" operator **A** (parameter `Aop`) and the size of its output (parameter `nAop`) for proper memory allocation.

(When I first looked at module `solver_reg` I was appalled by the many lines of code, especially all the declarations. Then I realized how much much worse was Fortran 77 where I needed to write a new solver for every pair of operators. This one solver module works for all operator pairs and for many optimization descent strategies because these "objects" are arguments. These more powerful objects require declarations that are more complicated than the simple objects of Fortran 77. As an author I have a dilemma: To make algorithms compact (and seem simple) requires many careful definitions. When these definitions put in the code, they are careful, but the code becomes annoyingly verbose. Otherwise, the definitions must go in the surrounding natural language where they are not easily made precise.)
solver_reg

After all the definitions, we load the negative of the data into the residual. If a starting model $\mathbf{m}_0$ is present, then we update the data part of the residual $\mathbf{r}_d =$

```
module solver_reg_mod{                          # 0 = W (F J m - d)
  use chain0_mod + solver_report_mod            # 0 =    A   m
  logical, parameter, private  :: T = .true., F = .false.
contains
  subroutine solver_reg( m,d, Fop, Aop, stepper, nAop, niter,eps &
  ,             Wop,Jop,m0,rm0,err,resd,resm,mmov,rmov,verb) {
    optional :: Wop,Jop,m0,rm0,err,resd,resm,mmov,rmov,verb
    interface { #-------------------------- begin definitions -----------
       integer function Fop(adj,add,m,d){real::m(:),d(:);logical,intent(in)::adj,add}
       integer function Aop(adj,add,m,d){real::m(:),d(:);logical,intent(in)::adj,add}
       integer function Wop(adj,add,m,d){real::m(:),d(:);logical,intent(in)::adj,add}
       integer function Jop(adj,add,m,d){real::m(:),d(:);logical,intent(in)::adj,add}
       integer function stepper(forget,m,g,rr,gg) {
         real, dimension(:)  ::        m,g,rr,gg
         logical             :: forget           }
    }
    real, dimension(:),    intent(in)          :: d, m0,rm0
    integer,               intent(in)          :: niter, nAop
    logical,               intent(in)          :: verb
    real,                  intent(in)          :: eps
    real, dimension(:),    intent(out)         :: m,err, resd,resm
    real, dimension(:,:),  intent(out)         ::        rmov,mmov
    real, dimension(size( m))                  :: g
    real, dimension(size( d) + nAop), target   :: rr, gg, tt
    real, dimension(:), pointer                :: rd, gd, td
    real, dimension(:), pointer                :: rm, gm, tm
    integer                                    :: iter, stat
    logical                                    :: forget
    rd => rr(1:size(d)); rm => rr(1+size(d):)
    gd => gg(1:size(d)); gm => gg(1+size(d):)
    td => tt(1:size(d)); tm => tt(1+size(d):)
    if(present(Wop)) stat=Wop(F,F,-d,rd) # begin initialization ---------
    else rd = -d                                          #Rd  = -W  d
    rm = 0.; if(present(rm0)) rm=rm0                      #Rm  =    Rm0
    if(present( m0)){ m=m0                                #m   =     m0
       if(present(Wop)) call chain0(Wop,Fop,F,T,m,rd,td)
       else             stat=           Fop(F,T,m,rd   )  #Rd +=  WF m0
       stat = Aop(F,T,eps*m0,rm)                          #Rm += e A m0
```

$\mathbf{Fm}_0 - \mathbf{d}$ and we load the model part of the residual $\mathbf{r}_m = \mathbf{Am}_0$. Otherwise we begin from a zero model $\mathbf{m}_0 = \mathbf{0}$ and thus the model part of the residual $\mathbf{r}_m$ is also zero. After this initialization, subroutine `solver_reg()` begins an iteration loop by first computing the proposed model perturbation $\Delta\mathbf{m}$ (called `g` in the program) with the adjoint operator:

$$\Delta\mathbf{m} \quad \longleftarrow \quad \begin{bmatrix} \mathbf{F}' & \mathbf{A}' \end{bmatrix} \begin{bmatrix} \mathbf{r}_d \\ \mathbf{r}_m \end{bmatrix} \tag{3.16}$$

Using this value of $\Delta\mathbf{m}$, we can find the implied change in residual $\Delta\mathbf{r}$ as

$$\Delta \begin{bmatrix} \mathbf{r}_d \\ \mathbf{r}_m \end{bmatrix} \quad \longleftarrow \quad \begin{bmatrix} \mathbf{F} \\ \mathbf{A} \end{bmatrix} \Delta\mathbf{m} \tag{3.17}$$

and the last thing in the loop is to use the optimization step function `stepper()` to choose the length of the step size and to choose how much of the previous step to include.

An example of using the new solver is subroutine `invint1`. I chose to implement the model roughening operator $\mathbf{A}$ with the convolution subroutine `tcai1()` /prog:tcai1, which has transient end effects (and an output length equal to the input

```
module invint {                         # invint -- INVerse INTerpolation in 1-D.
  use lint1
  use tcai1
  use cgstep_mod
  use solver_reg_mod
contains
  subroutine invint1( niter, coord, dd, o1, d1, aa, mm, eps, mmov) {
    integer,               intent (in)   :: niter               # iterations
    real,                  intent (in)   :: o1, d1, eps         # axis, scale
    real, dimension (:),   pointer       :: coord, aa           # aa is filter
    real, dimension (:),   intent (in)   :: dd                  # data
    real, dimension (:),   intent (out)  :: mm                  # model
    real, dimension (:,:), intent (out)  :: mmov                # movie
    integer                              :: nreg                # size of A m
    nreg = size( aa) + size( mm)                                # transient
    call lint1_init( o1, d1, coord )                            # interpolation
    call tcai1_init( aa)                                        # filtering
    call solver_reg( m=mm, d=dd, Fop=lint1_lop, stepper=cgstep, niter=niter, &
        Aop=tcai1_lop, nAop = nreg, eps = eps,mmov = mmov,verb=.true.)
    call cgstep_close( )
    }
}
```

Back

length plus the filter length). The adjoint of subroutine `tcai1()` suggests perturbations in the convolution input (not the filter). invint1

Figure 3.13 shows an example for a $(1, -2, 1)$ filter with $\epsilon = 1$. The continuous curve representing the model $\mathbf{m}$ passes through the data points. Because the models are computed with transient convolution end-effects, the models tend to damp linearly to zero outside the region where signal samples are given.

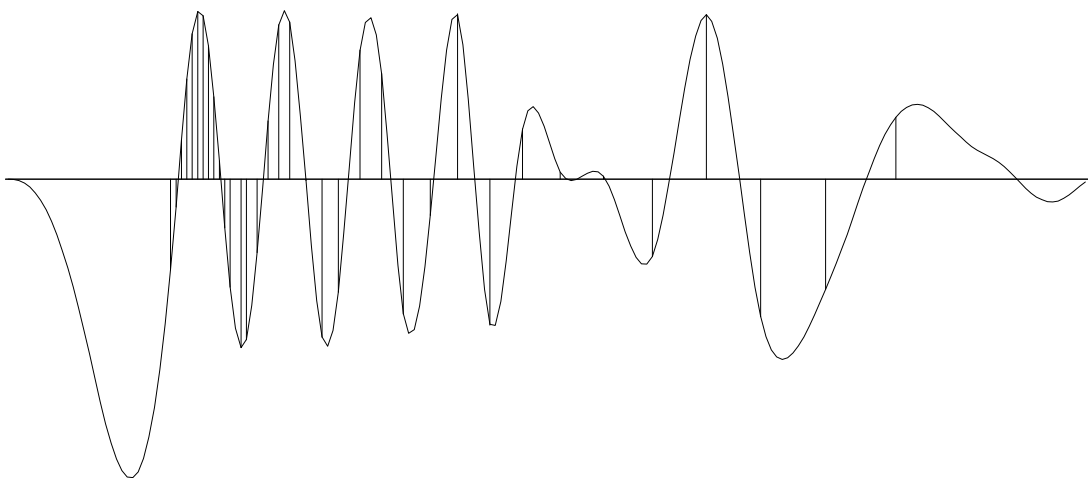


Figure 3.13: Sample points and estimation of a continuous function through them. iin-im1-2+190 [ER,M]

To show an example where the result is clearly a theoretical answer, I prepared another figure with the simpler filter $(1, -1)$. When we minimize energy in the first derivative of the waveform, the residual distributes itself uniformly between data

points so the solution there is a straight line. Theoretically it should be a straight line because a straight line has a vanishing second derivative, and that condition arises by differentiating by $\mathbf{x}'$, the minimized quadratic form $\mathbf{x}'\mathbf{A}'\mathbf{A}\mathbf{x}$, and getting $\mathbf{A}'\mathbf{A}\mathbf{x} = \mathbf{0}$. (By this logic, the curves between data points in Figure 3.13 must be cubics.) The $(1, -1)$ result is shown in Figure 3.14.

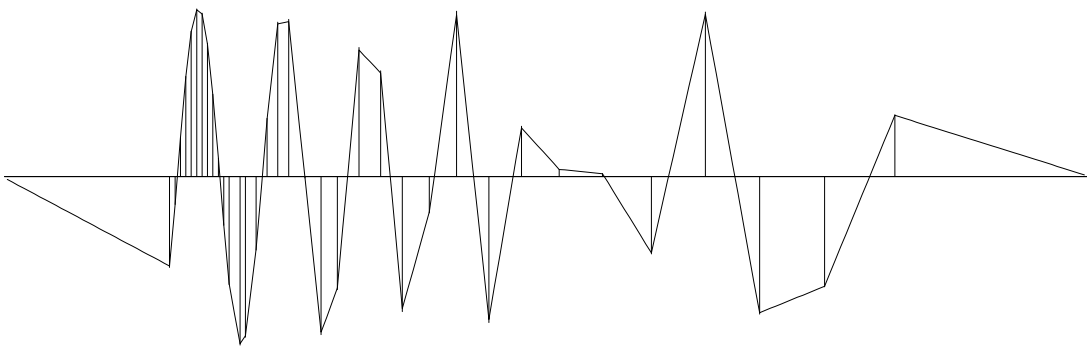


Figure 3.14: The same data samples and a function through them that minimizes the energy in the first derivative. iin-im1-1a90 [ER,M]

The example of Figure 3.14 has been a useful test case for me. You'll see it again in later chapters. What I would like to show you here is a movie showing the convergence to Figure 3.14. Convergence occurs rapidly where data points are close together. The large gaps, however, fill at a rate of one point per iteration.

3.4.1. Abandoned theory for matching wells and seismograms

Let us consider theory to construct a map $\mathbf{m}$ that fits dense seismic data $\mathbf{s}$ and the well data $\mathbf{w}$. The first goal $\mathbf{0} \approx \mathbf{Lm} - \mathbf{w}$ says that when we linearly interpolate from the map, we should get the well data. The second goal $\mathbf{0} \approx \mathbf{A}(\mathbf{m} - \mathbf{s})$ (where $\mathbf{A}$ is a roughening operator like ∇ or ∇^2) says that the map $\mathbf{m}$ should match the seismic data $\mathbf{s}$ at high frequencies but need not do so at low frequencies.

$$\begin{aligned} \mathbf{0} &\approx \mathbf{Lm} - \mathbf{w} \\ \mathbf{0} &\approx \mathbf{A}(\mathbf{m} - \mathbf{s}) \end{aligned} \tag{3.18}$$

Although (3.18) is the way I originally formulated the well-fitting problem, I abandoned it for several reasons: First, the map had ample pixel resolution compared to other sources of error, so I switched from linear interpolation to binning. Once I was using binning, I had available the simpler empty-bin approaches. These have the further advantage that it is not necessary to experiment with the relative weighting between the two goals in (3.18). A formulation like (3.18) is more likely to be helpful where we need to handle rapidly changing functions where binning is

inferior to linear interpolation, perhaps in reflection seismology where high resolution is meaningful.

EXERCISES:

1 It is desired to find a compromise between the Laplacian roughener and the gradient roughener. What is the size of the residual space?

2 Like the seismic prospecting industry, you have solved a huge problem using binning. You have computer power left over to do a few iterations with linear interpolation. How much does the cost per iteration increase? Should you refine your model mesh, or can you use the same model mesh that you used when binning?

3.5. PREJUDICE, BULLHEADEDNESS, AND CRO VALIDATION

First we first look at data **d**. Then we think about a model **m**, and an operator **L** to link the model and the data. Sometimes the operator is merely the first term in

a series expansion about $(\mathbf{m}_0, \mathbf{d}_0)$. Then we fit $\mathbf{d} - \mathbf{d}_0 \approx \mathbf{L}(\mathbf{m} - \mathbf{m}_0)$. To fit the model, we must reduce the fitting residuals. Realizing that the importance of a data residual is not always simply the size of the residual but is generally a function of it, we conjure up (topic for later chapters) a weighting function (which could be a filter) operator $\mathbf{W}$. This defines our data residual:

$$\mathbf{r}_d \quad = \quad \mathbf{W}[\mathbf{L}(\mathbf{m} - \mathbf{m}_0) \; - \; (\mathbf{d} - \mathbf{d}_0)] \tag{3.19}$$

Next we realize that the data might not be adequate to determine the model, perhaps because our comfortable dense sampling of the model ill fits our economical sparse sampling of data. Thus we adopt a fitting goal that mathematicians call "regularization" and we might call a "model style" goal or more simply, a quantification of our prejudice about models. We express this by choosing an operator $\mathbf{A}$, often simply a roughener like a gradient (the choice again a topic in this and later chapters). It defines our model residual by $\mathbf{Am}$ or $\mathbf{A}(\mathbf{m} - \mathbf{m}_0)$, say we choose

$$\mathbf{r}_m \quad = \quad \mathbf{Am} \tag{3.20}$$

In an ideal world, our model prejudice would not conflict with measured data, however, life is not so simple. Since conflicts between data and preconceived notions invariably arise (and they are why we go to the expense of acquiring data)

we need an adjustable parameter that measures our "bullheadedness", how much we intend to stick to our preconceived notions in spite of contradicting data. This parameter is generally called epsilon ϵ because we like to imagine that our bullheadedness is small. (In mathematics, ϵ is often taken to be an infinitesimally small quantity.) Although any bullheadedness seems like a bad thing, it must be admitted that measurements are imperfect too. Thus as a practical matter we often find ourselves minimizing

$$\min \quad := \quad \mathbf{r}_d \cdot \mathbf{r}_d \; + \; \epsilon^2 \, \mathbf{r}_m \cdot \mathbf{r}_m \tag{3.21}$$

and wondering what to choose for ϵ. I have two suggestions: My simplest suggestion is to choose ϵ so that the residual of data fitting matches that of model styling. Thus

$$\epsilon \quad = \quad \sqrt{\frac{\mathbf{r}_d \cdot \mathbf{r}_d}{\mathbf{r}_m \cdot \mathbf{r}_m}} \tag{3.22}$$

My second suggestion is to think of the force on our final solution. In physics, force is associated with a gradient. We have a gradient for the data fitting and another for

the model styling:

$$\mathbf{g}_d = \mathbf{L}'\mathbf{W}'\mathbf{r}_d \tag{3.23}$$

$$\mathbf{g}_m = \mathbf{A}'\mathbf{r}_m \tag{3.24}$$

We could balance these forces by the choice

$$\epsilon = \sqrt{\frac{\mathbf{g}_d \cdot \mathbf{g}_d}{\mathbf{g}_m \cdot \mathbf{g}_m}} \tag{3.25}$$

Although we often ignore ϵ in discussing the formulation of a problem, when time comes to solve the problem, reality intercedes. Generally, $\mathbf{r}_d$ has different physical units than $\mathbf{r}_m$ (likewise $\mathbf{g}_d$ and $\mathbf{g}_m$) and we cannot allow our solution to depend on the accidental choice of units in which we express the problem. I have had much experience choosing ϵ, but it is only recently that I boiled it down to the above two suggestions. Normally I also try other values, like double or half those of the above choices, and I examine the solutions for subjective appearance. If you find any insightful examples, please tell me about them.

Computationally, we could choose a new ϵ with each iteration, but it is more expeditious to freeze ϵ, solve the problem, recompute ϵ, and solve the problem

again. I have never seen a case where more than one iteration was necessary.

People who work with small problems (less than about 10^3 vector components) have access to an attractive theoretical approach called cross-validation. Simply speaking, we could solve the problem many times, each time omitting a different data value. Each solution would provide a model that could be used to predict the omitted data value. The quality of these predictions is a function of ϵ and this provides a guide to finding it. My objections to cross validation are two-fold: First, I don't know how to apply it in the large problems like we solve in this book (I should think more about it); and second, people who worry much about ϵ, perhaps first should think more carefully about their choice of the filters **W** and **A**, which is the focus of this book. Notice that both **W** and **A** can be defined with a scaling factor which is like scaling ϵ. Often more important in practice, with **W** and **A** we have a scaling factor that need not be constant but can be a function of space or spatial frequency within the data space and/or model space.

Chapter 4

The helical coordinate

For many years it has been true that our most powerful signal-analysis techniques are in *one*-dimensional space, while our most important applications are in *multi*-dimensional space. The helical coordinate system makes a giant step towards over-

coming this difficulty.

Many geophysical map estimation problems appear to be multidimensional, but actually they are not. To see the tip of the iceberg, consider this example: On a two-dimensional cartesian mesh, the function

0	0	0	0
0	1	1	0
0	1	1	0
0	0	0	0

has the autocorrelation

1	2	1
2	4	2
1	2	1

.

Likewise, on a one-dimensional cartesian mesh,
the function $b =$

1	1	0	0	$\cdots$	0	1	1

has the autocorrelation $r =$

1	2	1	0	$\cdots$	0	2	4	2	0	$\cdots$	1	2	1

.

Observe the numbers in the one-dimensional world are identical with the numbers in the two-dimensional world. This correspondence is no accident.

4.1. FILTERING ON A HELIX

Figure 4.1 shows some two-dimensional shapes that are convolved together. The left panel shows an impulse response function, the center shows some impulses, and the right shows the superposition of responses.

A surprising, indeed amazing, fact is that Figure 4.1 was not computed with a two-dimensional convolution program. It was computed with a one-dimensional computer program. It could have been done with anybody's one-dimensional convolution program, either in the time domain or in the fourier domain. This magical trick is done with the helical coordinate system.

A basic idea of filtering, be it in one dimension, two dimensions, or more, is that you have some filter coefficients and some sampled data; you pass the filter over the data; at each location you find an output by crossmultiplying the filter coefficients times the underlying data and summing the terms.

The helical coordinate system is much simpler than you might imagine. Ordinarily, a plane of data is thought of as a collection of columns, side by side. Instead, imagine the columns stored end-to-end, and then coiled around a cylinder. This is the helix. Fortran programmers will recognize that fortran's way of storing 2-D arrays in one-dimensional memory is exactly what we need for this helical mapping.

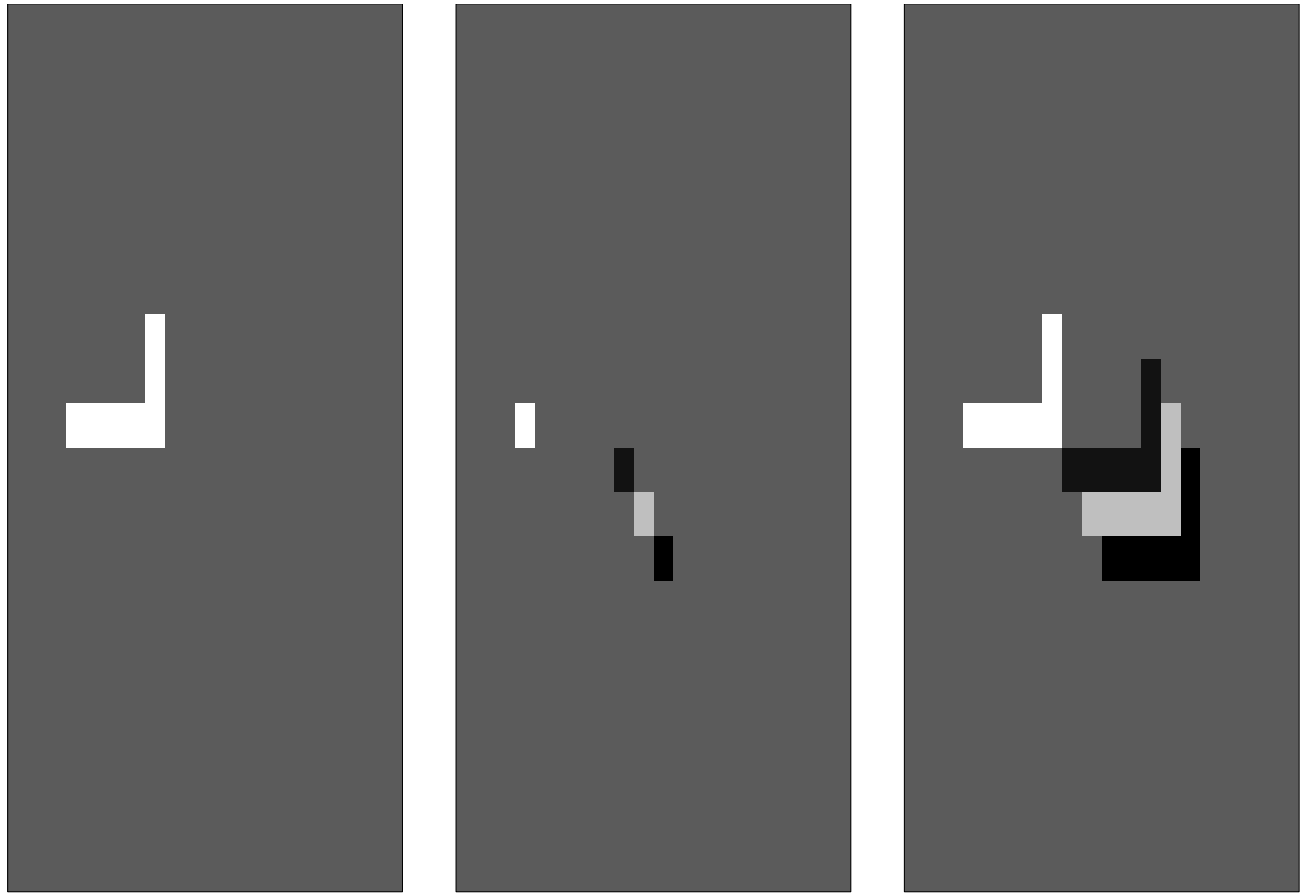


Figure 4.1: Two-dimensional convolution as performed in one dimension by module `helicon` hlx-diamond90 [ER]

Seismologists sometimes use the word "supertrace" to describe a collection of seismograms stored "end-to-end". Figure 4.2 shows a helical mesh for 2-D data on a cylinder. Darkened squares depict a 2-D filter shaped like the Laplacian operator $\partial_{xx} + \partial_{yy}$. The input data, the filter, and the output data are all on helical meshes all of which could be unrolled into linear strips. A compact 2-D filter like a Laplacian, on a helix is a sparse 1-D filter with long empty gaps.

Since the values output from filtering can be computed in any order, we can slide the filter coil over the data coil in any direction. The order that you produce the outputs is irrelevant. You could compute the results in parallel. We could, however, slide the filter over the data in the screwing order that a nut passes over a bolt. The screw order is the same order that would be used if we were to unwind the coils into one-dimensional strips and convolve them across one another. The same filter coefficients overlay the same data values if the 2-D coils are unwound into 1-D strips. The helix idea allows us to obtain the same convolution output in either of two ways, a one-dimensional way, or a two-dimensional way. I used the one-dimensional way to compute the obviously two-dimensional result in Figure 4.1.

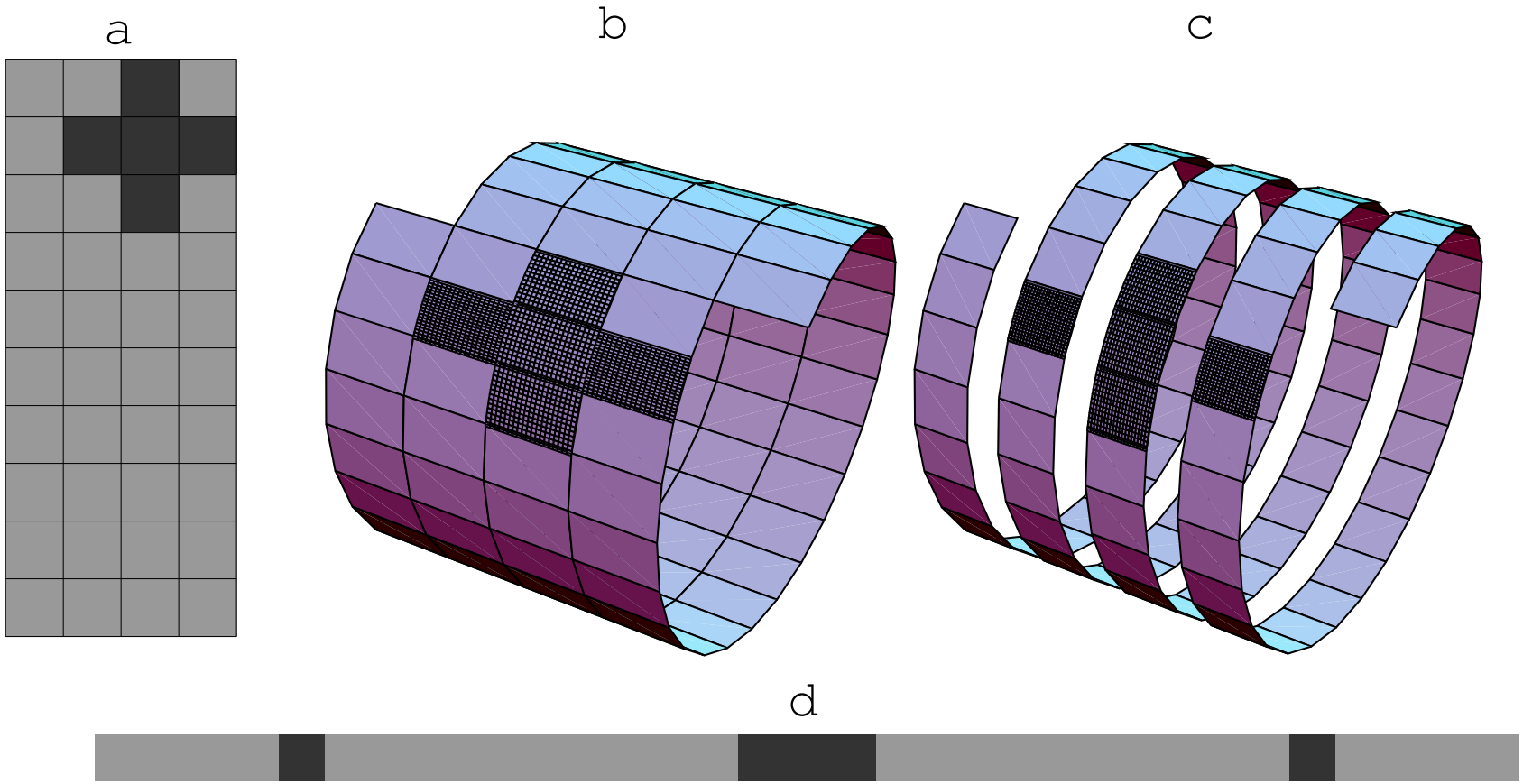


Figure 4.2: Filtering on a helix. The same filter coefficients overlay the same data values if the 2-D coils are unwound into 1-D strips. (*Mathematica* drawing by Sergey Fomel) hlx-sergey-helix [CR]

4.1.1. Review of 1-D recursive filters

Convolution is the operation we do on polynomial coefficients when we multiply polynomials. Deconvolution is likewise for polynomial division. Often these ideas are described as polynomials in the variable Z. Take $X(Z)$ to denote the polynomial whose coefficients are samples of input data, and let $A(Z)$ likewise denote the filter. The convention I adopt here is that the first coefficient of the filter has the value +1, so the filter's polynomial is $A(Z) = 1 + a_1 Z + a_2 Z^2 + \cdots$. To see how to convolve, we now identify the coefficient of Z^k in the product $Y(Z) = A(Z)X(Z)$. The usual case (k larger than the number N_a of filter coefficients) is

$$y_k = x_k + \sum_{i=1}^{N_a} a_i x_{k-i} \tag{4.1}$$

Convolution computes y_k from x_k whereas deconvolution (also called back substitution) does the reverse. Rearranging (4.1) we get

$$x_k = y_k - \sum_{i=1}^{N_a} a_i x_{k-i} \tag{4.2}$$

where now we are finding the output x_k from its past outputs x_{k-i} and from the present input y_k. We see that the deconvolution process is essentially the same as the convolution process, except that the filter coefficients are used with opposite polarity; and they are applied to the past *outputs* instead of the past *inputs*. That is why deconvolution must be done sequentially while convolution can be done in parallel.

4.1.2. Multidimensional deconvolution breakthrough

Deconvolution (polynomial division) can undo convolution (polynomial multiplication). A magical property of the helix is that we can consider 1-D convolution to be the same as 2-D convolution. Hence is a second magical property: We can use 1-D *de*convolution to undo convolution, whether that convolution was 1-D or 2-D. Thus, we have discovered how to undo 2-D convolution. We have discovered that 2-D deconvolution on a helix is equivalent to 1-D deconvolution. The helix enables us to do multidimensional deconvolution.

Deconvolution is recursive filtering. Recursive filter outputs cannot be computed in parallel, but must be computed sequentially as in one dimension, namely,

in the order that the nut screws on the bolt.

Recursive filtering sometimes solves big problems with astonishing speed. It can propagate energy rapidly for long distances. Unfortunately, recursive filtering can also be unstable. The most interesting case, near resonance, is also near instability. There is a large literature and extensive technology about recursive filtering in one dimension. The helix allows us to apply that technology to two (and more) dimensions. It is a huge technological breakthrough.

In 3-D we simply append one plane after another (like a 3-D fortran array). It is easier to code than to explain or visualize a spool or torus wrapped with string, etc.

4.1.3. Examples of simple 2-D recursive filters

Let us associate x- and y-derivatives with a finite-difference stencil or template. (For simplicity take $\Delta x = \Delta y = 1$.)

$$\frac{\partial}{\partial x} \quad = \quad \boxed{1 \mid -1} \tag{4.3}$$

$$\frac{\partial}{\partial y} \quad = \quad \begin{array}{|r|}\hline 1 \\ \hline -1 \\ \hline \end{array} \tag{4.4}$$

Convolving a data plane with the stencil (4.3) forms the x-derivative of the plane. Convolving a data plane with the stencil (4.4) forms the y-derivative of the plane. On the other hand, *deconvolving* with (4.3) integrates data along the x-axis for each y. Likewise, deconvolving with (4.4) integrates data along the y-axis for each x. Next we look at a fully two-dimensional operator (like the cross derivative ∂_{xy}).

A nontrivial two-dimensional convolution stencil is

$$\begin{array}{|r|l|}\hline 0 & -1/4 \\ \hline 1 & -1/4 \\ \hline -1/4 & -1/4 \\ \hline \end{array} \tag{4.5}$$

We will convolve and deconvolve a data plane with this operator. Although everything is shown on a plane, the actual computations are done in one dimension with equations (4.1) and (4.2). Let us manufacture the simple data plane shown on the left in Figure 4.3. Beginning with a zero-valued plane, we add in a copy of the filter (4.5) near the top of the frame. Nearby add another copy with opposite polarity.

Finally add some impulses near the bottom boundary. The second frame in Figure 4.3 is the result of deconvolution by the filter (4.5) using the one-dimensional equation (4.2). Notice that deconvolution turns the filter itself into an impulse, while it turns the impulses into comet-like images. The use of a helix is evident by the comet images wrapping around the vertical axis. The filtering in Figure 4.3 ran along a helix from left to right. Figure 4.4 shows a second filtering running from right to left. Filtering in the reverse direction is the adjoint. After deconvolving both ways, we have accomplished a symmetical smoothing. The final frame undoes the smoothing to bring us exactly back to where we started. The smoothing was done with two passes of *deconvolution* and it is undone by two passes of *convolution*. No errors, no evidence remains of any of the boundaries where we have wrapped and truncated.

Chapter 5 explains the important practical role to be played by a multidimensional operator for which we know the exact inverse. Other than multidimensional Fourier transformation, transforms based on polynomial multiplication and division on a helix are the only known easily invertible linear operators.

In seismology we often have occasion to steer summation along beams. Such an impulse response is shown in Figure 4.6. Of special interest are filters that destroy

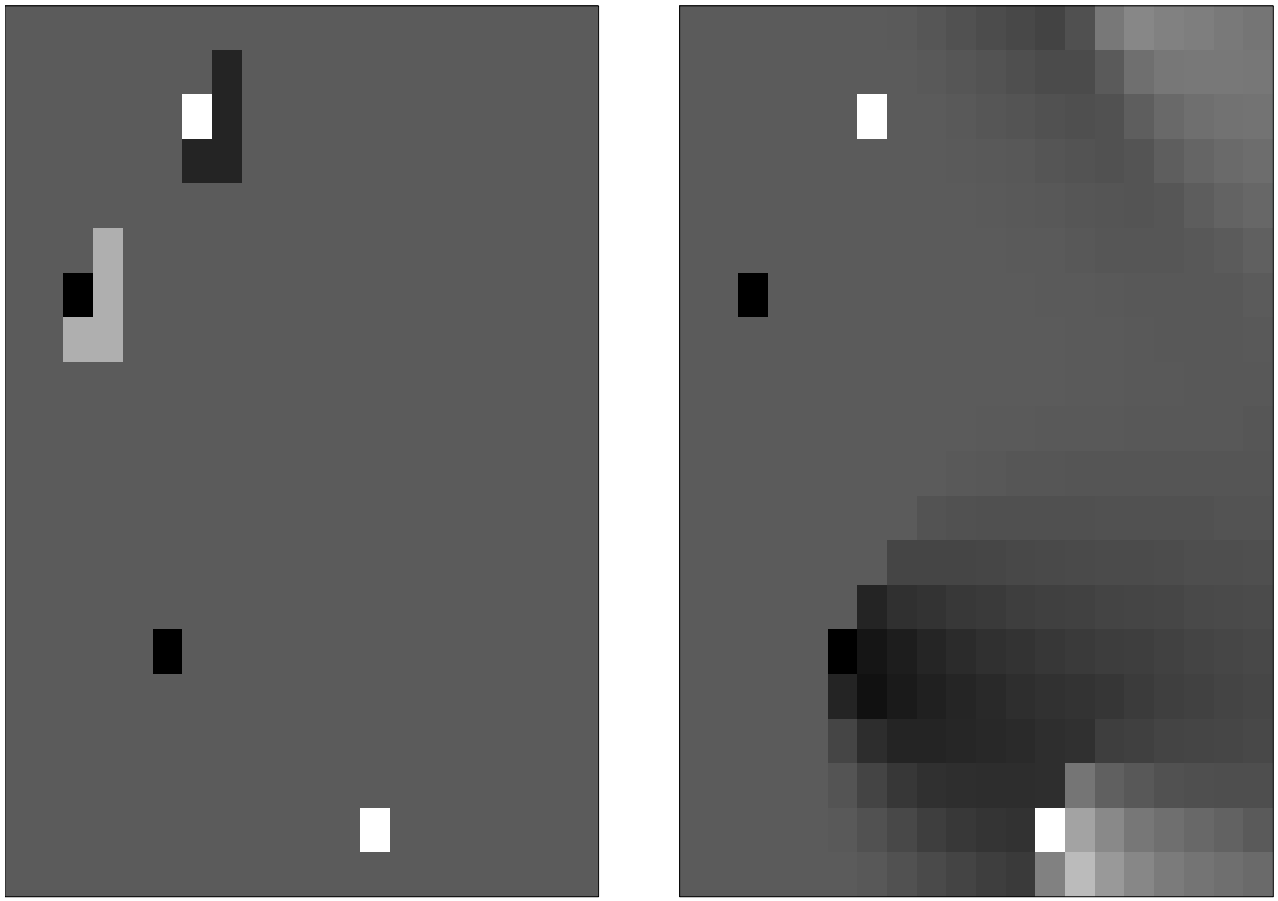


Figure 4.3: Illustration of 2-D deconvolution. Left is the input. Right is after deconvolution with the filter (4.5) as preformed by by module `polydiv` hlx-wrap90 [ER]

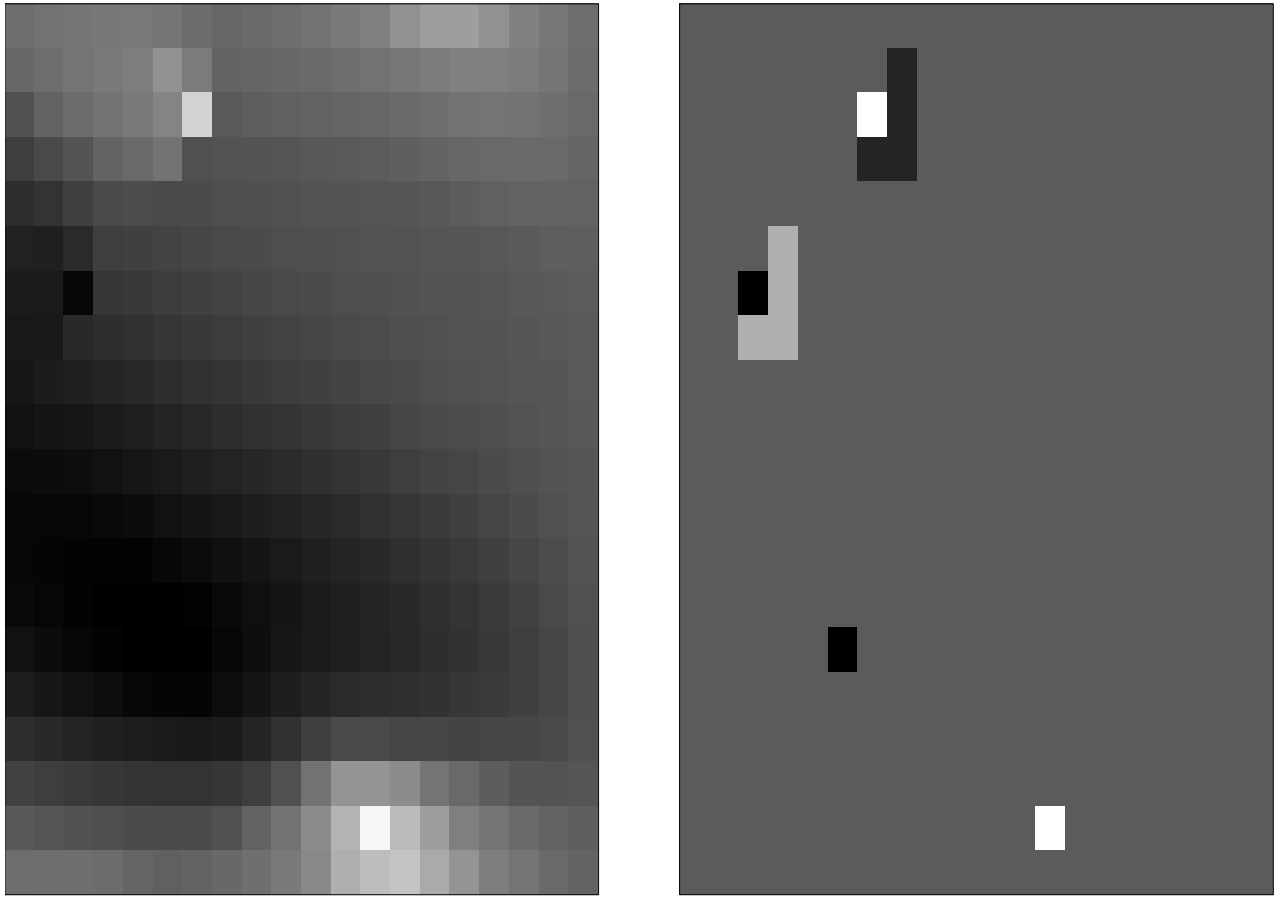
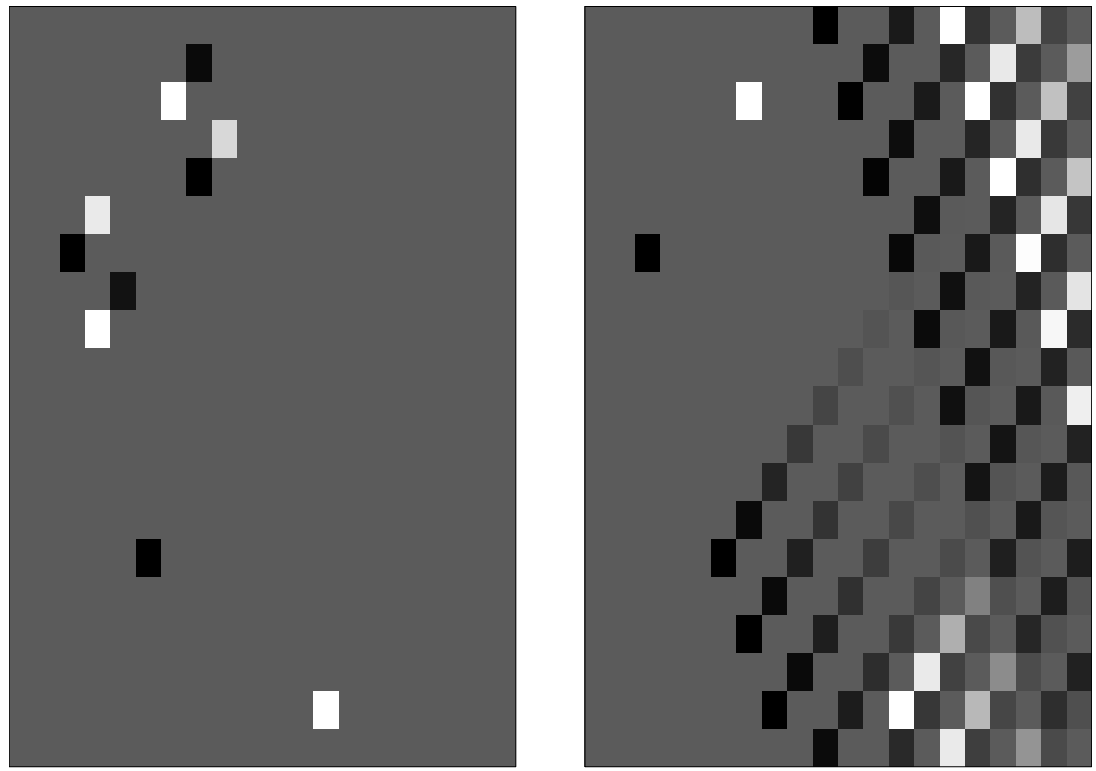


Figure 4.4: Recursive filtering backwards (leftward on the space axis) is done by the *adjoint* of 2-D deconvolution. Here we see that 2-D *deconvolution* compounded with its adjoint is exactly inverted by 2-D *convolution* and its adjoint. hlx-hback90 [ER]

Figure 4.5: A simple low-order 2-D filter whose inverse contains plane waves of two different dips. One of them is spatially aliased. hlx-waves90 [ER]



plane waves. The inverse of such a filter creates plane waves. Such filters are like wave equations. A filter that creates two plane waves is illustrated in figure 4.5.

4.1.4. Coding multidimensional de/convolution

Let us unroll the filter helix seen in Figure 4.2 and see what we have. Start from the idea that a 2-D filter is generally made from a cluster of values near one another in two dimensions similar to the Laplacian operator in the figure. We see that in the helical approach, a 2-D filter is a 1-D filter containing some long intervals of zeros. The intervals are about the length of a 1-D seismogram.

Our program for 2-D convolution with a 1-D convolution program, could convolve with the somewhat long 1-D strip, but it is much more cost effective to ignore the many zeros, which is what we do. We do not multiply by the backside zeros, nor do we even store them in memory. Whereas an ordinary convolution program would do time shifting by a code line like `iy=ix+lag`, Module `helicon` /prog:helicon ignores the many zero filter values on backside of the tube by using the code `iy=ix+lag(ia)` where a counter `ia` ranges over the nonzero filter coefficients. Before operator `helicon` is invoked, we need to prepare two lists, one list

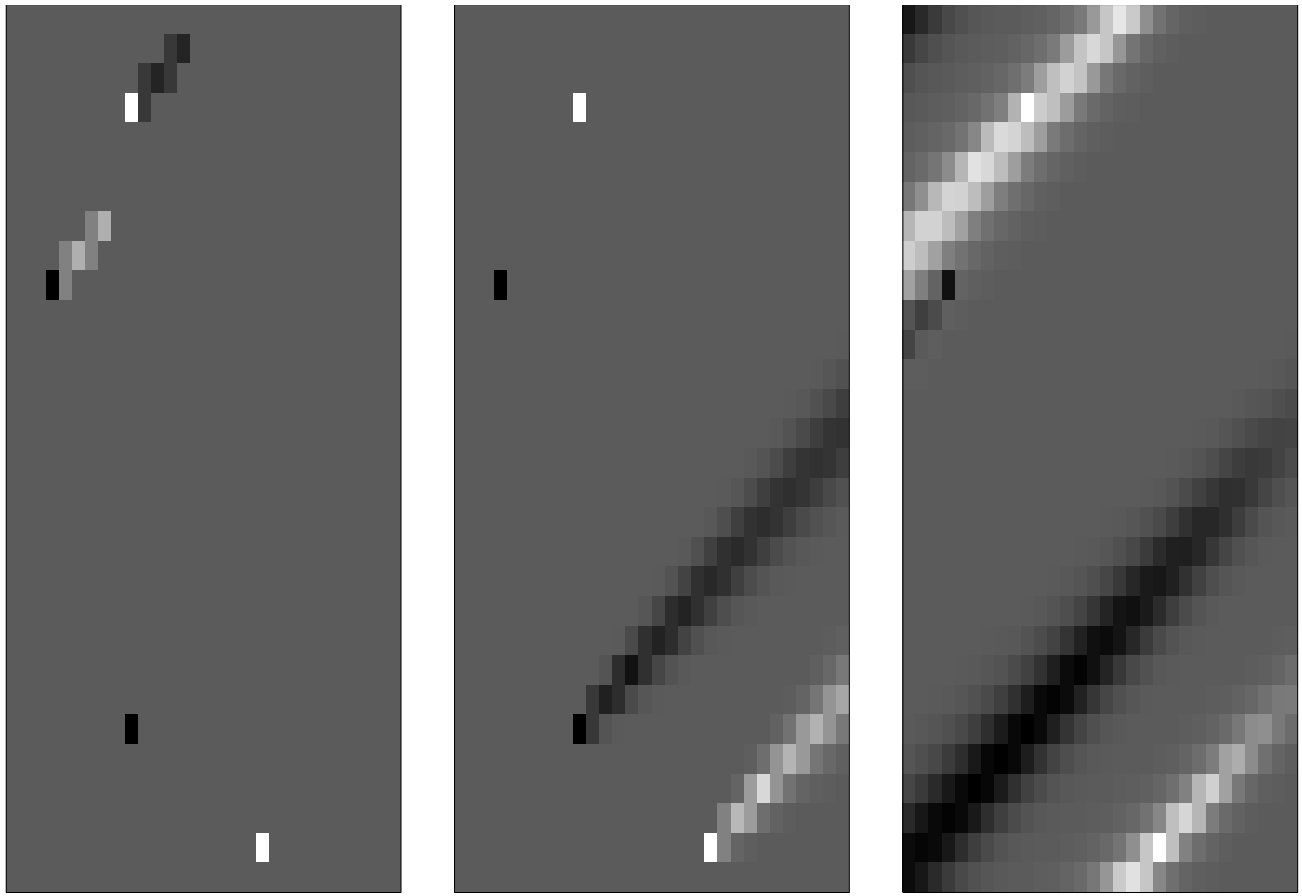


Figure 4.6: A simple low-order 2-D filter whose inverse times its inverse adjoint, is approximately a dipping seismic arrival. hlx-dip90 [ER]

containing nonzero filter coefficients `flt(ia)`, and the other list containing the corresponding lags `lag(ia)` measured to include multiple wraps around the helix. For example, the 2-D Laplace operator can be thought of as the 1-D filter

1	0	$\cdots$	0	1	-4	1	0	$\cdots$	0	1

$\xrightarrow{\text{helical boundaries}}$

	1	
1	-4	1
	1	

(4.6)

The first filter coefficient in equation (4.6) is $+1$ as implicit to module `helicon`. To apply the Laplacian on a 1000×1000 mesh requires the filter inputs:

```
 i     lag(i)    flt(i)
---    ------    -----
 1       999       1
 2      1000      -4
 3      1001       1
 4      2000       1
```

Here we choose to use "declaration of a type", a modern computer language feature that is absent from Fortran 77. Fortran 77 has the built in complex arithmetic

```
module helix {                                  # DEFINE helix filter type
  type filter {
    real,    dimension( :), pointer :: flt      # (nh) filter coefficients
    integer, dimension( :), pointer :: lag      # (nh) filter lags
    logical, dimension( :), pointer :: mis      # (nd) boundary conditions
  }
contains
  subroutine allocatehelix( aa, nh ) {          # allocate a filter
    type( filter) :: aa
    integer       :: nh                         # count of filter coefs (excl 1)
    allocate( aa%flt( nh), aa%lag( nh))         # allocate filter and lags.
    nullify( aa%mis)                            # set null pointer for "mis".
    aa%flt = 0.                                 # zero filter coef values
  }
  subroutine deallocatehelix( aa) {             # destroy a filter
    type( filter) :: aa
    deallocate( aa%flt, aa%lag)                 # free memory
    if( associated( aa%mis))                    # if logicals were allocated
        deallocate( aa%mis)                     # free them
  }
}
```

Back

type. In module `helix` we define a type `filter`, actually, a helix filter. After making this definition, it will be used by many programs. The helix filter consists of three vectors, a real valued vector of filter coefficients, an integer valued vector of filter lags, and an optional vector that has logical values "`.TRUE.`" for output locations that will not be computed (either because of boundary conditions or because of missing inputs). The filter vectors are the size of the nonzero filter coefficents (excluding the leading 1.) while the logical vector is long and relates to the data size. The `helix` module allocates and frees memory for a helix filter. By default, the logical vector is not allocated but is set to `null` with the `nullify` operator and ignored. helix

For those of you with no Fortran 90 experience, the "`%`" appearing in the helix module denotes a pointer. Fortran 77 has no pointers (or everything is a pointer). The C, C++, and Java languages use "`.`" to denote pointers. C and C++ also have a second type of pointer denoted by "`->`". The behavior of pointers is somewhat different in each language. Never-the-less, the idea is simple. In module `helicon` /prog:helicon you see the expression `aa%flt(ia)`. It refers to the filter named `aa`. Any filter defined by the `helix` module contains three vectors, one of which is named `flt`. The second component of the `flt` vector in the `aa` filter is referred to as `aa%flt(2)` which in the example above refers to the value 4.0 in the center of

the laplacian operator. For data sets like above with 1000 points on the 1-axis, this value 4.0 occurs after 1000 lags, thus `aa%lag(2)=1000`.

Our first convolution operator `tcai1` /prog:tcai1 was limited to one dimension and a particular choice of end conditions. With the helix and Fortran 90 pointers, the operator `helicon` /prog:helicon is a *multidimensional* filter with considerable flexibility (because of the `mis` vector) to work around boundaries and missing data. helicon The code fragment `aa%lag(ia)` corresponds to `b-1` in `tcai1` /prog:tcai1.

Operator `helicon` did the convolution job for Figure 4.1. As with `tcai1` /prog:tcai1 the adjoint of filtering is filtering backwards which means unscrewing the helix.

The companion to convolution is deconvolution. The module `polydiv` /prog:polydiv is essentially the same as `polydiv1` /prog:polydiv1, but here it was coded using our new `filter` type in module `helix` /prog:polydiv1 which will simplify our many future uses of convolution and deconvolution. Although convolution allows us to work around missing input values, deconvolution does not (any input affects all subsequent outputs), so `polydiv` never references `aa%mis(ia)`. polydiv

```
module helicon {                          # Convolution, inverse to deconvolution.
#             Requires the filter be causal with an implicit "1." at the onset.
use helix
type( filter) :: aa
#%  _init( aa)
#%  _lop ( xx,  yy)
integer iy, ix, ia
if( adj)                # zero lag
        xx += yy
else
        yy += xx
do ia = 1, size( aa%lag) {
      do iy = 1  + aa%lag( ia), size( yy) {
            if( associated( aa%mis)) { if( aa%mis( iy)) cycle}
            ix = iy - aa%lag( ia)
            if( adj)
                        xx(ix) += yy(iy) * aa%flt(ia)
            else
                        yy(iy) += xx(ix) * aa%flt(ia)
            }
      }
}
```

Back

```
module polydiv {                                # Helix polynomial division
use helix
integer                                 :: nd
type( filter)                           :: aa
real, dimension (nd), allocatable :: tt
#%  _init ( nd, aa)
#%  _lop  ( xx, yy)
integer  ia, ix, iy
tt = 0.
if( adj) {
        do ix= nd, 1, -1 {
                tt( ix) = yy( ix)
                do ia = 1, size( aa%lag) {
                        iy = ix + aa%lag( ia);      if( iy > nd)  next
                        tt( ix) -=  aa%flt( ia) * tt( iy)
                        }
                }
        xx += tt
 } else {
        do iy= 1, nd {
                tt( iy) = xx( iy)
                do ia = 1, size( aa%lag) {
                        ix = iy - aa%lag( ia);      if( ix < 1)  next
                        tt( iy) -=  aa%flt( ia) * tt( ix)
                        }
                }
        yy += tt
        }
}
```

Back

EXERCISES:

1 Observe the matrix matrix (1.4) which corresponds to subroutine `tcai1` /prog:tcai1

What is the matrix corresponding to `helicon` /prog:helicon?

4.1.5. Causality in two-dimensions

In one dimension, most filters of interest have a short memory. Significant filter coefficients are concentrated shortly after $t = 0$. The favorite example in Physics is the damped harmonic oscillator, all of which is packed into a two-lag filter (second order differential equation). The complete story is rich in mathematics and in concepts, but to sum up, filters fall into two categories according to the numerical values of their coefficients. There are filters for which equations (4.1) and (4.2) work as desired and expected. These filters are called "minimum phase". There are also filters for which (4.2) is a disaster numerically, the feedback process diverging to infinity.

Divergent cases correspond to physical processes that require boundary conditions. Equation (4.2) only allows for initial conditions. I oversimplify by trying to collapse an entire book (FGDP) into a few sentences by saying here that for any

fixed spectrum there exist many filters. Of these, only one has stable polynomial division. That filter has its energy compacted as soon as possible after the "1.0" at zero lag.

Now let us turn to two dimensions. Filters of interest will correspond to energy concentrated near the end of a helix. Let us examine the end of a helix. At the very end, as in the 1-D case, is a coefficient with the numerical value 1.0. Keeping only coefficients within two mesh points in any direction from the 1.0, we copy the coefficients from near the end of the helix to a cartesian mesh like this:

$$\begin{matrix} h & c & 0 \\ p & d & 0 \\ q & e & \mathbf{1} \\ s & f & a \\ u & g & b \end{matrix} \quad = \quad \begin{matrix} h & c & \cdot \\ p & d & \cdot \\ q & e & \cdot \\ s & f & a \\ u & g & b \end{matrix} \quad + \quad \begin{matrix} \cdot & \cdot & 0 \\ \cdot & \cdot & 0 \\ \cdot & \cdot & \mathbf{1} \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{matrix} \tag{4.7}$$

$$2-\text{D filter} \quad = \quad \text{variable} \quad + \quad \text{constrained}$$

where $a, b, c, ..., u$ are adjustable coefficients.

Which side of the little rectangular patch of coefficients we choose to place the

1.0 is rather arbitrary. The important matter is that as a matter of principle, the 1.0 is expected to lie along one side of the little patch. It is rarely (if ever) found at a corner of the patch. It is important that beyond the 1.0 (in whatever direction that may be) the filter coefficients must be zero because in one dimension, these coefficients lie before zero lag. Our foundations, the basic convolution-deconvolution pair (4.1) and (4.2) are applicable only to filters with all coefficients *after* zero lag.

Time-series analysis is rich with concepts that the helix now allows us to apply to many dimensions. First is the notion of an impulse function. Observe that an impulse function on the 2-D surface of the helical cylinder maps to an impulse function on the 1-D line of the unwound coil. An autocorrelation function that is an impulse corresponds both to a white (constant) spectrum in 1-D and to a white (constant) spectrum in 2-D. Next we look at a particularly important autocorrelation function and see how 2-D is the same as 1-D.

4.2. FINITE DIFFERENCES ON A HELIX

The function

$$r \quad = \quad \boxed{-1 \mid 0 \mid \cdots \mid 0 \mid -1 \mid 4 \mid -1 \mid 0 \mid \cdots \mid 0 \mid -1} \tag{4.8}$$

is an autocorrelation function. It is symmetrical about the "4" and its Fourier transform is positive for all frequencies. Digging out our old textbooks[1] we discover how to compute a causal wavelet with this autocorrelation. I used the "Kolmogoroff spectral-factorization method" to find this wavelet h:

$$h \quad = \quad \boxed{1.791 \mid -.651 \mid -.044 \mid -.024 \mid \cdots \mid \cdots \mid -.044 \mid -.087 \mid -.200 \mid -.55} \tag{4.9}$$

According to the Kolmogoroff theory, if we form the autocorrelation of h, we will get r. This is not obvious from the numbers themselves because the computation requires a little work.

---

[1] PVI or FGDP, for example, explain spectral factorization. More concisely in PVI, more richly in FGDP.

Let the time reversed version of $\hbar$ be denoted $\hbar'$. This notation is consistant with an idea from Chapter 1 that the adjoint of a filter matrix is another filter matrix with a reversed filter. In engineering it is conventional to use the asterisk symbol "$*$" to denote convolution. Thus, the idea that the autocorrelation of a signal $\hbar$ is a convolution of the signal $\hbar$ with its time reverse (adjoint) can be written as $\hbar' * \hbar = \hbar * \hbar' = r$.

Wind the signal r around a vertical-axis helix to see its two-dimensional shape $\mathcal{R}$:

$$r \quad \xrightarrow{\text{helical boundaries}} \quad \begin{array}{|c|c|c|} \hline & -1 & \\ \hline -1 & 4 & -1 \\ \hline & -1 & \\ \hline \end{array} \quad = \quad \mathcal{R} \tag{4.10}$$

This 2-D filter is the negative of the finite-difference representation of the Laplacian operator, generally denoted $\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$. Now for the magic: Wind the signal $\hbar$

around the same helix to see its two-dimensional shape $\mathcal{H}$

$$\mathcal{H} \quad = \quad \begin{array}{|c|c|c|c|c|c|c|c|c|} \hline & & & & 1.791 & -.651 & -.044 & -.024 & \cdots \\ \hline \cdots & -.044 & -.087 & -.200 & -.558 & & & & \\ \hline \end{array} \tag{4.11}$$

In the representation (4.11) we see the coefficients diminishing rapidly away from maximum value 1.791. My claim is that the two-dimensional autocorrelation of (4.11) is (4.10). You verified this idea earlier when the numbers were all ones. You can check it again in a few moments if you drop the small values, say 0.2 and smaller.

Since the autocorrelation of $\mathcal{H}$ is $\mathcal{H}' * \mathcal{H} = \mathcal{R} = -\nabla^2$ is a second derivative, the operator $\mathcal{H}$ must be something like a first derivative. As a geophysicist, I found it natural to compare the operator $\frac{\partial}{\partial y}$ with $\mathcal{H}$ by applying them to a local topographic map. The result shown in Figure 4.7 is that $\mathcal{H}$ enhances drainage patterns whereas $\frac{\partial}{\partial y}$ enhances mountain ridges.

The operator $\mathcal{H}$ has curious similarities and differences with the familiar gradient and divergence operators. In two-dimensional physical space, the gradient maps one field to *two* fields (north slope and east slope). The factorization of $-\nabla^2$ with

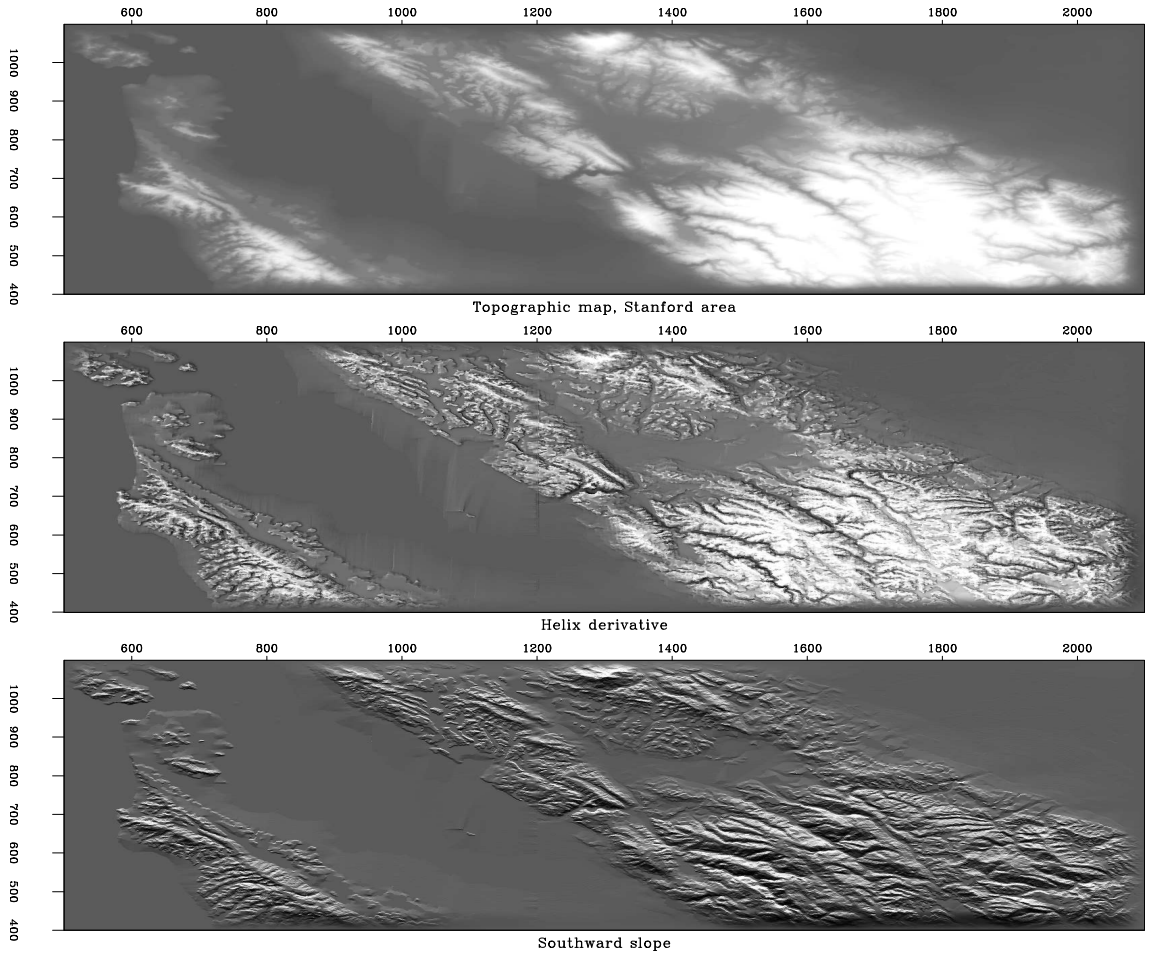


Figure 4.7: Topography, helical derivative, slope south. hlx-helocut90 [ER,M]

the helix gives us the operator $\mathcal{H}$ that maps one field to *one* field. Being a one-to-one transformation (unlike gradient and divergence) the operator $\mathcal{H}$ is potentially invertible by deconvolution (recursive filtering).

I have chosen the name[2] "helix derivative" or "helical derivative" for the operator $\mathcal{H}$. A telephone pole has a narrow shadow behind it. The helix integral (middle frame of Figure 4.8) and the helix derivative (left frame) show shadows with an angular bandwidth approaching 180°.

Our construction makes $\mathcal{H}$ have the energy spectrum $k_x^2 + k_y^2$, so the magnitude of the Fourier transform is $\sqrt{k_x^2 + k_y^2}$. It is a cone centered and with value zero at the origin. By contrast, the components of the ordinary gradient have amplitude responses $|k_x|$ and $|k_y|$ that are lines of zero across the (k_x, k_y)-plane.

The rotationally invariant cone in the Fourier domain contrasts sharply with the nonrotationally invariant function shape in (x, y)-space. The difference must arise

---

[2] Any fact this basic should be named in some earlier field of mathematics or theoretical physics. Admittedly, the concept exists on an infinite cartesian plane without a helix, but all my codes in a finite space involve the helix, and the helix concept led me to it.

from the phase spectrum. The factorization (4.11) is nonunique in that causality associated with the helix mapping can be defined along either x- or y-axes; thus the operator (4.11) can be rotated or reflected.

This is where the story all comes together. One-dimensional theory, either the old Kolmogoroff spectral factorization, or the new Wilson-Burg spectral-factorization method produces not merely a causal wavelet with the required autocorrelation. It produces one that is stable in deconvolution. Using $\mathcal{H}$ in one-dimensional polynomial division, we can solve many formerly difficult problems very rapidly. Consider the Laplace equation with sources (Poisson's equation). Polynomial division and its reverse (adjoint) gives us $p = (q/\mathcal{H})/\mathcal{H}'$ which means that we have solved $\nabla^2 p = -q$ by using polynomial division on a helix. Using the seven coefficients shown, the cost is fourteen multiplications (because we need to run both ways) per mesh point. An example is shown in Figure 4.8.

Figure 4.8 contains both the helix derivative and its inverse. Contrast them to the x- or y-derivatives (doublets) and their inverses (axis-parallel lines in the (x, y)-plane). Simple derivatives are highly directional whereas the helix derivative is only slightly directional achieving its meagre directionality entirely from its phase spectrum.

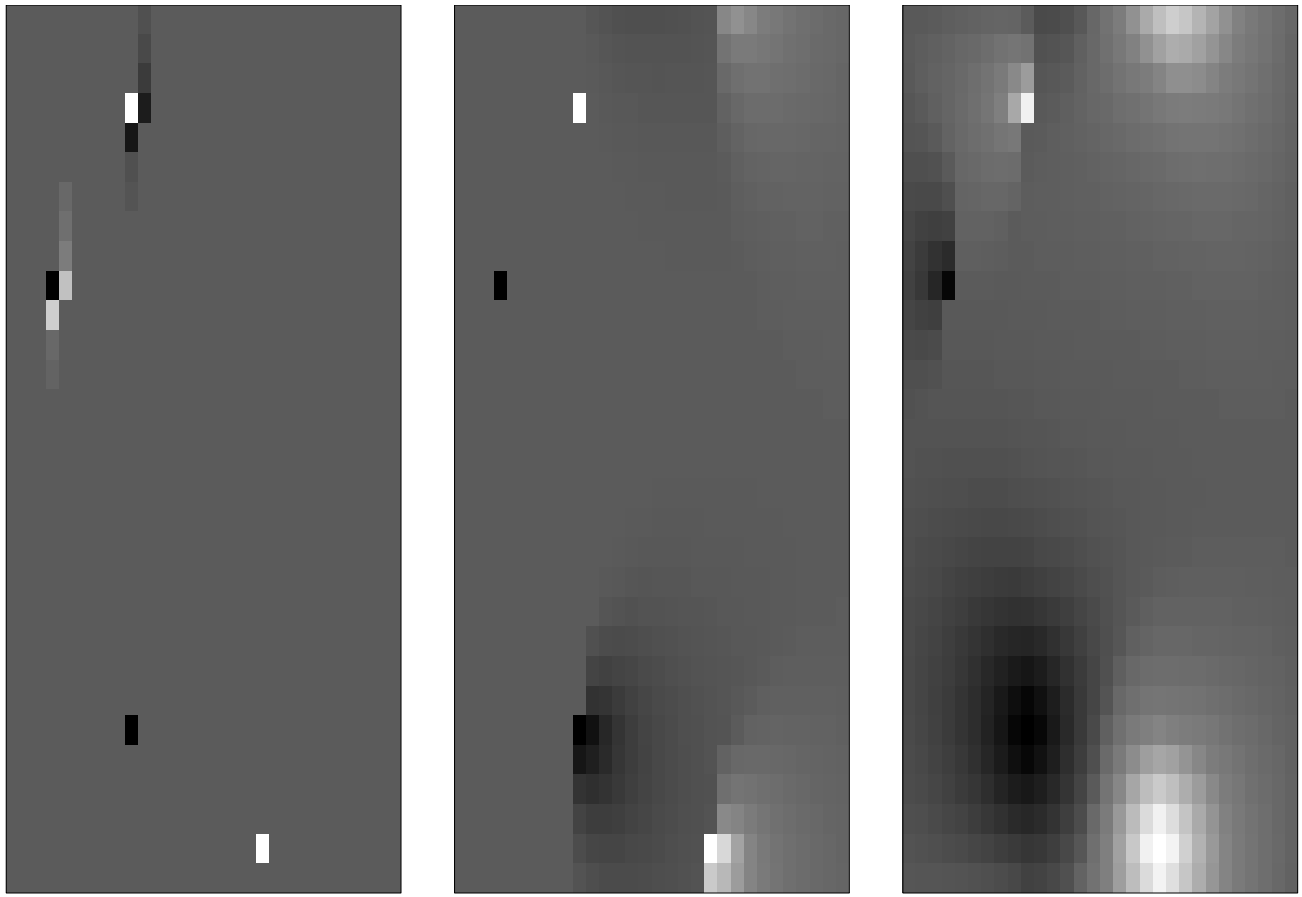


Figure 4.8: Deconvolution by a filter whose autocorrelation is the two-dimensional Laplacian operator. Amounts to solving the Poisson equation. Left is q; Middle is $q/\mathcal{H}$; Right is $(q/\mathcal{H})/\mathcal{H}'$. hlx-lapfac90 [ER]

In practice we often require an isotropic filter. Such a filter is a function of $k_r = \sqrt{k_x^2 + k_y^2}$. It could be represented as a sum of helix derivatives to integer powers.

4.2.1. Matrix view of the helix

Physics on a helix can be viewed thru the eyes of matrices and numerical analysis. This is not easy because the matrices are so huge. Discretize the (x, y)-plane to an $N \times M$ array and pack the array into a vector of $N \times M$ components. Likewise pack the Laplacian operator $\partial_{xx} + \partial_{yy}$ into a matrix. For a 4×3 plane, that matrix is

shown in equation (4.12).

$$
-\nabla^2 \quad = \quad \left[\begin{array}{cccc|cccc|cccc}
4 & -1 & \cdot & \cdot & -1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
-1 & 4 & -1 & \cdot & \cdot & -1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & -1 & 4 & -1 & \cdot & \cdot & -1 & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & -1 & 4 & h & \cdot & \cdot & -1 & \cdot & \cdot & \cdot & \cdot \\
\hline
-1 & \cdot & \cdot & h & 4 & -1 & \cdot & \cdot & -1 & \cdot & \cdot & \cdot \\
\cdot & -1 & \cdot & \cdot & -1 & 4 & -1 & \cdot & \cdot & -1 & \cdot & \cdot \\
\cdot & \cdot & -1 & \cdot & \cdot & -1 & 4 & -1 & \cdot & \cdot & -1 & \cdot \\
\cdot & \cdot & \cdot & -1 & \cdot & \cdot & -1 & 4 & h & \cdot & \cdot & -1 \\
\hline
\cdot & \cdot & \cdot & \cdot & -1 & \cdot & \cdot & h & 4 & -1 & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & -1 & \cdot & \cdot & -1 & 4 & -1 & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & -1 & \cdot & \cdot & -1 & 4 & -1 \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & -1 & \cdot & \cdot & -1 & 4
\end{array}\right] \tag{4.12}
$$

The two-dimensional matrix of coefficients for the Laplacian operator is shown in (4.12), where, on a cartesian space, $h = 0$, and in the helix geometry, $h = -1$. (A similar partitioned matrix arises from packing a cylindrical surface into a 4×3 ar-

ray.) Notice that the partitioning becomes transparent for the helix, $h = -1$. With the partitioning thus invisible, the matrix simply represents one-dimensional convolution and we have an alternative analytical approach, one-dimensional Fourier Transform. We often need to solve sets of simultaneous equations with a matrix similar to (4.12). The method we use is triangular factorization.

Although the autocorrelation r has mostly zero values, the factored autocorrelation a has a great number of nonzero terms, but fortunately they seem to be converging rapidly (in the middle) so truncation (of the middle coefficients) seems reasonable. I wish I could show you a larger matrix, but all I can do is to pack the

signal a into shifted columns of a lower triangular matrix $\mathbf{A}$ like this:

$$\mathbf{A} = \begin{bmatrix}
1.8 & . & . & . & . & . & . & . & . & . & . & . \\
-.6 & 1.8 & . & . & . & . & . & . & . & . & . & . \\
0.0 & -.6 & 1.8 & . & . & . & . & . & . & . & . & . \\
-.2 & 0.0 & -.6 & 1.8 & . & . & . & . & . & . & . & . \\
-.6 & -.2 & 0.0 & -.6 & 1.8 & . & . & . & . & . & . & . \\
. & -.6 & -.2 & 0.0 & -.6 & 1.8 & . & . & . & . & . & . \\
. & . & -.6 & -.2 & 0.0 & -.6 & 1.8 & . & . & . & . & . \\
. & . & . & -.6 & -.2 & 0.0 & -.6 & 1.8 & . & . & . & . \\
. & . & . & . & -.6 & -.2 & 0.0 & -.6 & 1.8 & . & . & . \\
. & . & . & . & . & -.6 & -.2 & 0.0 & -.6 & 1.8 & . & . \\
. & . & . & . & . & . & -.6 & -.2 & 0.0 & -.6 & 1.8 & . \\
. & . & . & . & . & . & . & -.6 & -.2 & 0.0 & -.6 & 1.8
\end{bmatrix} \tag{4.13}$$

If you will allow me some truncation approximations, I now claim that the laplacian represented by the matrix in equation (4.12) is factored into two parts $-\nabla^2 = \mathbf{A}'\mathbf{A}$ which are upper and lower triangular matrices whose product forms the autocorre-

lation seen in (4.12). Recall that triangular matrices allow quick solutions of simultaneous equations by backsubstitution. That is what we do with our deconvolution program.

4.3. CAUSALITY AND SPECTAL FACTORIZATION

Mathematics sometimes seems a mundane subject, like when it does the "accounting" for an engineer. Other times it brings unexpected amazing new concepts into our lives. This is the case with the study of causality and spectral factorization. There are many little-known, amazing, fundamental ideas here I would like to tell you about. We won't get to the bottom of any of them but it's fun and useful to see what they are and how to use them.

Start with an example. Consider a mechanical object. We can strain it and watch it stress or we can stress it and watch it strain. We feel knowledge of the present and past stress history is all we need to determine the present value of strain. Likewise, the converse, history of strain should tell us the stress. We could say there

is a filter that takes us from stress to strain; likewise another filter takes us from strain to stress. What we have here is a pair of filters that are mutually inverse under convolution. In the Fourier domain, one is literally the inverse of the other. What is remarkable is that in the time domain, both are causal. They both vanish before zero lag $\tau = 0$.

Not all causal filters have a causal inverse. The best known name for one that does is "minimum-phase filter." Unfortunately, this name is not suggestive of the fundamental property of interest, "causal with a causal (convolutional) inverse." I could call it CwCI. An example of a causal filter without a causal inverse is the unit delay operator — with Z-transforms, the operator Z itself. If you delay something, you can't get it back without seeing into the future, which you are not allowed to do. Mathematically, $1/Z$ cannot be expressed as a polynomial (actually, a convergent infinite series) in positive powers of Z.

Physics books don't tell us where to expect to find transfer functions that are CwCI. I think I know why they don't. Any causal filter has a "sharp edge" at zero time lag where it switches from nonresponsiveness to responsiveness. The sharp edge might cause the spectrum to be large at infinite frequency. If so, the inverse filter is small at infinite frequency. Either way, one of the two filters is unmanageable

with Fourier transform theory which (you might have noticed in the mathematical fine print) requires signals (and spectra) to have finite energy which means the function must get real small in that immense space on the t-axis and the ω axis. It is impossible for a function to be small and its inverse be small. These imponderables get more managable in the world of Time Series Analysis (discretized time axis).

4.3.1. The spectral factorization concept

Interesting questions arise when we are given a spectrum and find ourselves asking how to find a filter that has that spectrum. Is the answer unique? We'll see not. Is there always an answer that is causal? Almost always, yes. Is there always an answer that is causal with a causal inverse (CwCI)? Almost always, yes.

Let us have an example. Consider a filter like the familiar time derivative $(1,-1)$ except let us downweight the -1 a tiny bit, say $(1,-\rho)$ where $0 << \rho < 1$. Now the filter $(1,-\rho)$ has a spectrum $(1-\rho Z)(1-\rho/Z)$ with autocorrelation coefficients $(-\rho, 1+\rho^2, -\rho)$ that look a lot like a second derivative, but it is a tiny bit bigger in the middle. Two different waveforms, $(1,-\rho)$ and its time reverse both have the same autocorrelation. Spectral factorization could give us both $(1,-\rho)$ and

$(\rho, -1)$ but we always want the one that is CwCI. The bad one is weaker on its first pulse. Its inverse is not causal. Below are two expressions for the filter inverse to $(\rho, -1)$, the first divergent (filter coefficients at infinite lag are infinitely strong), the second convergent but noncausal.

$$\frac{1}{\rho - Z} = \frac{1}{\rho}\left(1 + Z/\rho + Z^2/\rho^2 + \cdots\right) \tag{4.14}$$

$$\frac{1}{\rho - Z} = \frac{-1}{Z}\left(1 + \rho/Z + \rho^2/Z^2 + \cdots\right) \tag{4.15}$$

(Please multiply each equation by $\rho - Z$ and see it reduce to $1 = 1$).

So we start with a power spectrum and we should find a CwCI filter with that energy spectrum. If you input to the filter an infinite sequence of random numbers (white noise) you should output something with the original power spectrum.

We easily inverse Fourier transform the square root of the power spectrum getting a symmetrical time function, but we need a function that vanishes before $\tau = 0$. On the other hand, if we already had a causal filter with the correct spectrum we could manufacture many others. To do so all we need is a family of delay operators to convolve with. A pure delay filter does not change the spectrum of anything.

Same for frequency-dependent delay operators. Here is an example of a frequency-dependent delay operator: First convolve with (1,2) and then deconvolve with (2,1). Both these have the same amplitude spectrum so their ratio has a unit amplitude (and nontrivial phase). If you multiply $(1+2Z)/(2+Z)$ by its Fourier conjugate (replace Z by $1/Z$) the resulting spectrum is 1 for all ω.

Anything whose nature is delay is death to CwCI. The CwCI has its energy as close as possible to $\tau = 0$. More formally, my first book, FGDP, proves that the CwCI filter has for all time τ more energy between $t = 0$ and $t = \tau$ than any other filter with the same spectrum.

Spectra can be factorized by an amazingly wide variety of techniques, each of which gives you a different insight into this strange beast. They can be factorized by factoring polynomials, by inserting power series into other power series, by solving least squares problems, by taking logarithms and exponentials in the Fourier domain. I've coded most of them and still find them all somewhat mysterious.

Theorems in Fourier analysis can be interpreted physically in two different ways, one as given, the other with time and frequency reversed. For example, convolution in one domain amounts to multiplication in the other. If we were to express the CwCI concept with reversed domains, instead of saying the "energy comes as

quick as possible after $\tau = 0$" we would say "the frequency function is as close to $\omega = 0$ as possible." In other words, it is minimally wiggly with time. Most applications of spectral factorization begin with a spectrum, a real, positive function of frequency. I once achieved minor fame by starting with a real, positive function of space, a total magnetic field $\sqrt{H_x^2 + H_z^2}$ measured along the x-axis and I reconstructed the magnetic field components H_x and H_z that were minimally wiggly in space.

4.3.2. Cholesky decomposition

Conceptually the simplest computational method of spectral factorization might be "Cholesky decomposition." For example, the matrix of (4.13) could have been found by Cholesky factorization of (4.12). The Cholesky algorithm takes a positive-definite matrix $\mathbf{Q}$ and factors it into a triangular matrix times its transpose, say $\mathbf{Q} = \mathbf{T}'\mathbf{T}$.

It is easy to reinvent the Cholesky factorization algorithm. To do so, simply write all the components of a 3×3 triangular matrix $\mathbf{T}$ and then explicitly multiply these elements times the transpose matrix $\mathbf{T}'$. You will find that you have everything

you need to recursively build the elements of $\mathbf{T}$ from the elements of $\mathbf{Q}$. Likewise for a 4×4 matrix, etc.

The 1×1 case shows that the Cholesky algorithm requires square roots. Matrix elements are not always numbers. Sometimes they are polynomials such as Z-transforms. To avoid square roots there is a variation of the Cholesky method. In this variation, we factor $\mathbf{Q}$ into $\mathbf{Q} = \mathbf{T}'\mathbf{D}\mathbf{T}$ where $\mathbf{D}$ is a diagonal matrix.

Once a matrix has been factored into upper and lower triangles, solving simultaneous equations is simply a matter of two backsubstitutions: (We looked at a special case of backsubstitution with equation (1.23).) For example, we often encounter simultaneous equations of the form $\mathbf{B}'\mathbf{B}\mathbf{m} = \mathbf{B}'\mathbf{d}$. Suppose the positive-definite matrix $\mathbf{B}'\mathbf{B}$ has been factored into triangle form $\mathbf{T}'\mathbf{T}\mathbf{m} = \mathbf{B}'\mathbf{d}$. To find $\mathbf{m}$ we first backsolve $\mathbf{T}'\mathbf{x} = \mathbf{B}'\mathbf{d}$ for the vector $\mathbf{x}$. Then we backsolve $\mathbf{T}\mathbf{m} = \mathbf{x}$. When $\mathbf{T}$ happens to be a band matrix, then the first backsubstitution is filtering down a helix and the second is filtering back up it. Polynomial division is a special case of back substitution.

Poisson's equation $\nabla^2 \mathbf{p} = -\mathbf{q}$ requires boundary conditions which we can honor when we filter starting from both ends. We cannot simply solve Poisson's equation as an initial-value problem. We could insert the laplace operator into the polynomial

division program, but the solution would diverge.

Being a matrix method, the Cholesky method of factorization has a cost proportional to the cube of the size of the matrix. Because our problems are very large and because the Cholesky method does not produce a useful result if we stop part way to completion, we look further. The Cholesky method is a powerful method but it does more than we require. The Cholesky method does not require band matrices, yet these matrices are what we very often find in applications, so we seek methods that take advantage of the special properties of band matrices.

4.3.3. Toeplitz methods

Band matrices are often called Toeplitz matrices. In the subject of Time Series Analysis are found spectral factorization methods that require computations proportional to the dimension of the matrix squared. They can often be terminated early with a reasonable partial result. Two Toeplitz methods, the Levinson method and the Burg method are described in my first textbook, FGDP. Our interest is multidimensional data sets so the matrices of interest are truely huge and the cost of Toeplitz methods is proportional to the square of the matrix size. Thus, before we find Toeplitz meth-

ods especially useful, we may need to find ways to take advantage of the sparsity of our filters.

4.3.4. Kolmogoroff spectral factorization

With Fourier analysis we find a method of spectral factorization that is as fast as Fourier transformation, namely $N \log N$ for a matrix of size N. This is very appealing. An earlier version of this book included such an algorithm. Pedagogically, I didn't like it in this book because it requires lengthy off-topic discussions of Fourier analysis which are already found in both my first book FGDP and my third book PVI.

The Kolmogoroff calculation is based on the logarithm of the spectrum. The logarithm of zero is minus infinity — an indicator that perhaps we cannot factorize a spectrum which becomes zero at any frequency. Actually, the logarithmic infinity is the gentlest kind. The logarithm of the smallest nonzero value in single precision arithmetic is about -36 which might not ruin your average calculation. Mathematicians have shown that the integral of the logarithm of the spectrum must be bounded so that some isolated zero values of the spectrum are not a disaster. In other words,

we can factor the (negative) second derivative to get the first derivative. This suggests we will never find a causal bandpass filter. It is a contradiction to desire both causality and a spectral band of zero gain.

The weakness of the Kolmogoroff method is related to its strength. Fourier methods strictly require the matrix to be a band matrix. A problem many people would like to solve is how to handle a matrix that is "almost" a band matrix — a matrix where any band changes slowly with location.

4.3.5. Blind deconvolution

A area of applications that leads directly to spectral factorization is "blind deconvolution." Here we begin with a signal. We form its spectrum and factor it. We could simply inspect the filter and interpret it, or we might deconvolve it out from the original data. This topic deserves a fuller exposition, say for example as defined in some of my earlier books. Here we inspect a novel example that incorporates the helix.

Solar physicists have learned how to measure the seismic field of the sun surface. If you created an impulsive explosion on the surface of the sun, what would

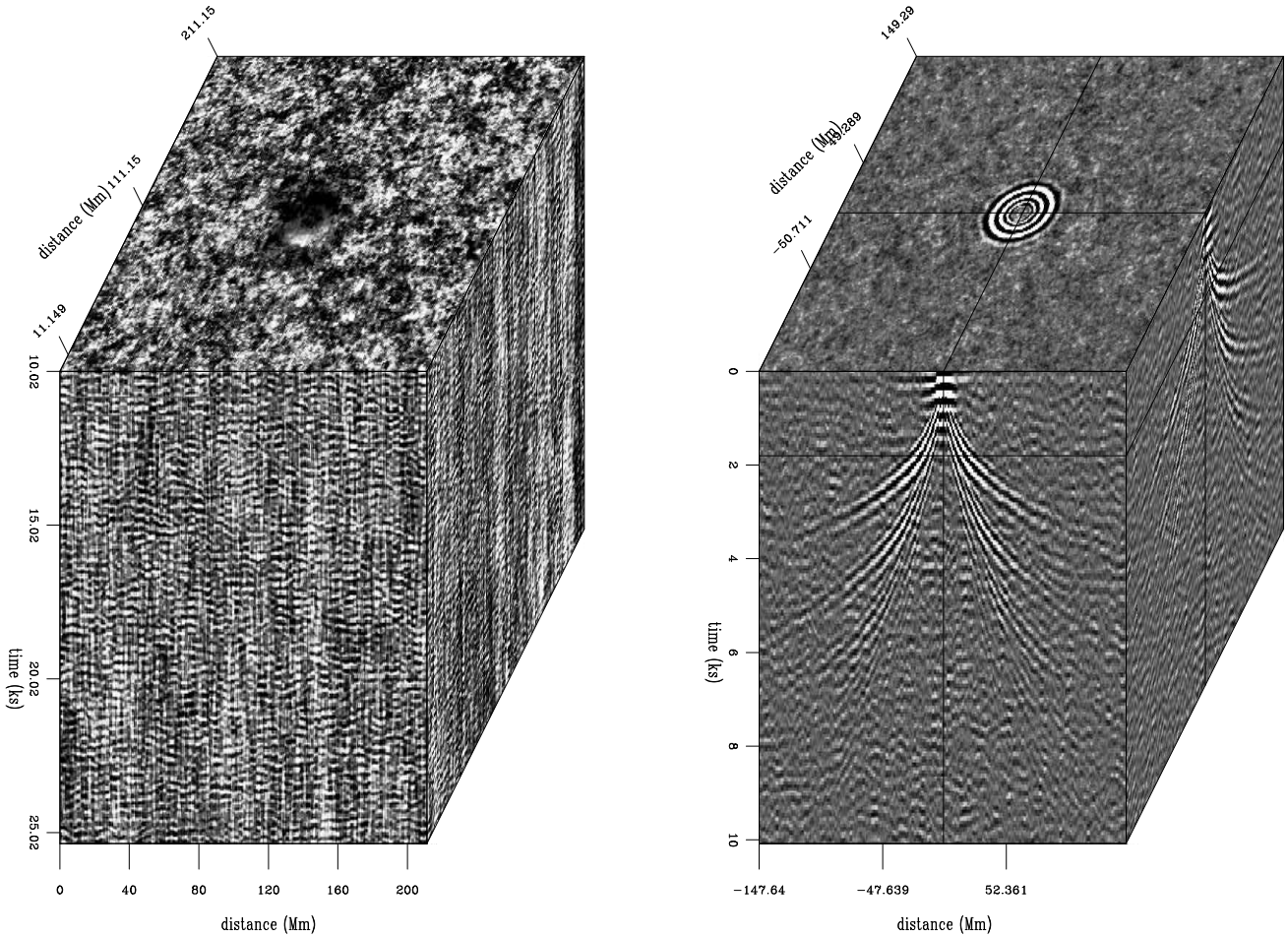


Figure 4.9: Raw seismic data on the sun (left). Impulse response of the sun (right) derived by Helix-Kolmogoroff spectral factorization. hlx-solar [NR,M]

the response be? James Rickett and I applied the helix idea along with Kolmogoroff spectral factorization to find the impulse response of the sun. Figure 4.9 shows a raw data cube and the derived impulse response. The sun is huge so the distance scale is in megameters (Mm). The United States is 5 Mm wide. Vertical motion of the sun is measured with a video-camera like device that measures vertical motion by doppler shift. From an acoustic/seismic point of view, the surface of the sun is a very noisy place. The figure shows time in kiloseconds (Ks). We see about 15 cycles in 5 Ks which is 1 cycle in about 333 sec. Thus the sun seems to oscillate vertically with about a 5 minute period. The top plane of the raw data in Figure 4.9 (left panel) happens to have a sun spot in the center. The data analysis here is not affected by the sun spot so please ignore it.

The first step of the data processing is to transform the raw data to its spectrum. With the helix assumption, computing the spectrum is virtually the same thing in 1-D space as in 3-D space. The resulting spectrum was passed to Kolmogoroff spectral factorization code. The resulting impulse response is on the right side of Figure 4.9. The plane we see on the right top is not lag time $\tau = 0$; it is lag time $\tau = 2$ Ks. It shows circular rings, as ripples on a pond. Later lag times (not shown) would be the larger circles of expanding waves. The front and side planes show tent-

like shapes. The slope of the tent gives the (inverse) velocity of the wave (as seen on the surface of the sun). The horizontal velocity we see on the sun surface turns out (by Snell's law) to be the same as that at the bottom of the ray. On the front face at early times we see the low velocity (steep) wavefronts and at later times we see the faster waves. This is because the later arrivals reach more deeply into the sun. Look carefully, and you can see two (or even three!) tents inside one another. These "inside tents" are the waves that have bounced once (or more!) from the surface of the sun. When a ray goes down and back up to the sun surface, it reflects and takes off again with the same ray shape. The result is that a given slope on the traveltime curve can be found again at twice the distance at twice the time.

4.4. WILSON-BURG SPECTRAL FACTORIZATION

(If you are new to this material, you should pass over this section.) Spectral factorization is the job of taking a power spectrum and from it finding a causal (zero before zero time) filter with that spectrum. Methods for this task (there are many)

not only produce a causal wavelet, but they typically produce one whose convolutional inverse is also causal. (This is called the "minimum phase" property.) In other words, with such a filter we can do stable deconvolution. Here I introduce a new method of spectral factorization that looks particularly suitable for the task at hand. I learned this new method from John Parker Burg who attributes it to an old paper by Wilson (I find Burg's explanation, below, much clearer than Wilson's.)

Below find subroutine `lapfac()` which was used in the previous section to factor the Laplacian operator. To invoke the factorization subroutine, you need to supply one side of an autocorrelation function. For example, let us specify the negative of the 2-D Laplacian (an autocorrelation) in a vector `n` $= 256 \times 256$ points long.

```
rr(1)     =  4.
rr(2)     = -1.
rr(1+256) = -1.
```

Subroutine `lapfac()` finds the helical derivative (factored negative Laplacian) and then prepares the required filter coefficient tables for the helix convolution and deconvolution subroutines. lapfac Subroutine `lapfacn()` has its main job done by subroutine `wilson_factor()` /prog:wilson shown after the Wilson-Burg theory.

```
module lapfac {                                 # Factor 2-D Laplacian.
  use wilson
contains
  function lapfac2( eps, n1, na)   result (aa) {
    type( filter)                               :: aa, lap
    real,                    intent( in) :: eps
    integer,                 intent( in) :: n1, na
    integer                                     :: i
    real                                        :: a0, lap0
    call allocatehelix( lap, 2)                 # laplacian filter
    lap0 = 4. + eps                             # zero lag coeff.
    lap%lag = (/ 1, n1 /)                       # lag(1)= 1; lag(2)=n1  # one side only
    lap%flt = -1.                               # flt(1)=-1; flt(2)=-1
    call allocatehelix( aa, 2*na)               # laplacian derivative
    aa%flt = 0.;                                # probably done already in allocation.
    do i = 1, na {
       aa%lag( i   ) =      i                   # early lags (first row)
       aa%lag( na+i) = n1 + i - na              # late lags (second row)
       }
    call wilson_init( 10 * n1 )
    call wilson_factor( 20, lap0, lap, a0, aa)
    call wilson_close()
    call deallocatehelix( lap)
    }
}
```

Back

4.4.1. Wilson-Burg theory

Newton's iteration for square roots

$$a_{t+1} \quad = \quad \frac{1}{2}\left(a_t + \frac{s}{a_t}\right) \tag{4.16}$$

converges quadratically starting from any real initial guess a_0 except zero. When a_0 is negative, Newton's iteration converges to the negative square root.

Quadratic convergence means that the square of the error $a_t - \sqrt{s}$ at one iteration is proportional to the error at the next iteration

$$a_{t+1} - \sqrt{s} \quad \sim \quad (a_t - \sqrt{s})^2 \quad = \quad a_t^2 - 2a_t\sqrt{s} + s \quad > \quad 0 \tag{4.17}$$

so, for example if the error is one significant digit at one iteration, at the next iteration it is two digits, then four, etc. We cannot use equation (4.17) in place of the Newton iteration itself, because it uses the answer $\sqrt{s}$ to get the answer a_{t+1}, and also we need the factor of proportionality. Notice, however, if we take the factor to be $1/(2a_t)$, then $\sqrt{s}$ cancels and equation (4.17) becomes itself the Newton iteration (4.16).

Another interesting feature of the Newton iteration is that all iterations (except possibly the initial guess) are above the ultimate square root. This is obvious from equation (4.17).

We can insert spectral functions in the Newton square-root iteration, for example $s(\omega)$ and $a(\omega)$. Where the first guess a_0 happens to match $\sqrt{s}$, it will match $\sqrt{s}$ at all iterations. The Newton iteration is

$$2\,\frac{a_{t+1}}{a_t} \quad = \quad 1 + \frac{s}{a_t^2} \tag{4.18}$$

Something inspires Wilson to express the spectrum $S = \bar{A}A$ as a Z-transform and then write the iteration

$$\frac{\bar{A}_{t+1}(1/Z)}{\bar{A}_t(1/Z)} + \frac{A_{t+1}(Z)}{A_t(Z)} \quad = \quad 1 + \frac{S(Z)}{\bar{A}_t(1/Z)\,A_t(Z)} \tag{4.19}$$

Now we are ready for the algorithm: Compute the right side of (4.19) by polynomial division forwards and backwards and then add 1. Then abandon negative lags and take half of the zero lag. Now you have $A_{t+1}(Z)/A_t(Z)$. Multiply out (convolve) the denominator $A_t(Z)$, and you have the desired result $A_{t+1}(Z)$. Iterate as long as you wish.

(Parenthetically, for those people familiar with the idea of minimum phase (if not, see FGDP or PVI), we show that $A_{t+1}(Z)$ is minimum phase: Both sides of (4.19) are positive, as noted earlier. Both terms on the right are positive. Since the Newton iteration always overestimates, the 1 dominates the rightmost term. After masking off the negative powers of Z (and half the zero power), the right side of (4.19) adds two wavelets. The 1/2 is wholly real, and hence its real part always dominates the real part of the rightmost term. Thus (after masking negative powers) the wavelet on the right side of (4.19) has a positive real part, so the phase cannot loop about the origin. This wavelet multiplies $A_t(Z)$ to give the final wavelet $A_{t+1}(Z)$ and the product of two minimum-phase wavelets is minimum phase.)

The input of the program is the spectrum $S(Z)$ and the output is the factor $A(Z)$, a function with the spectrum $S(Z)$. I mention here that in later chapters of this book, the factor $A(Z)$ is known as the inverse Prediction-Error Filter (PEF). In the Wilson-Burg code below, $S(Z)$ and $A(Z)$ are Z-transform polynomials but their lead coefficients are extracted off, so for example, $A(z) = (a_0) + (a_1 Z + a_2 Z^2 + \cdots)$ is broken into the two parts `a0` and `aa`. wilson

```
module wilson {
# Wilson's factorization
  use helicon
  use polydiv
  integer,                              private :: n
  real, dimension( :), allocatable, private :: auto, bb, cc, b, c
contains
  subroutine wilson_init( nmax) {
    integer, intent (in) :: nmax;    n = nmax
    allocate ( auto( 2*n-1), bb( 2*n-1), cc( 2*n-1), b(n), c(n))
    }
  subroutine wilson_factor( niter, s0, ss, a0, aa, verb) {
    integer,       intent( in)  :: niter # Newton iterations
    real,          intent( in)  :: s0    # autocorrelation zero lag
    type( filter), intent( in)  :: ss    # autocorrelation, other lags
    real,          intent( out) :: a0    # factor, zero lag
    type( filter)               :: aa    # factor, other lags
    logical,       intent( in)  :: verb
    optional                    :: verb
    real                        :: eps
    integer                     :: i, stat
    auto = 0.;  auto( n) = s0;  b( 1) =1.       # initialize
    auto( n+ss%lag) =  ss%flt                   # autocorrelation
    auto( n-ss%lag) =  ss%flt                   # symmetrize input auto.
    call helicon_init( aa)                      # multiply polynoms
    call polydiv_init( 2*n-1, aa)               # divide   polynoms
    do i = 1, niter {
       stat= polydiv_lop(.false.,.false., auto, bb)  # bb = S/A
       stat= polydiv_lop( .true.,.false., cc,   bb)  # cc = S/(AA')
       b( 2:n) = 0.5*( cc( n+1:2*n-1  ) +
                       cc( n-1:1    :-1)) / cc( n)     # b = plusside(1+cc)
       eps = maxval( abs( b(2:n)));   # "L1 norm"
       if (present (verb)) { if (verb) write (0,*) i, eps}
       stat= helicon_lop( .false., .false., b, c)      # c = A b
       aa%flt = c( 1+aa%lag)                           # put on helix
       if( eps < epsilon( a0))  break                  # convergence
       }
    a0 = sqrt( cc( n))
```

EXERCISES:

1 You hear from three different people that a more isotropic representation of the Laplacian is minus one sixth of

$$\begin{matrix} -1 & -4 & -1 \\ -4 & 20 & -4 \\ -1 & -4 & -1 \end{matrix}$$

What changes need to be made to subroutine `lapfac()`?

2 Fomel's factorization: A simple trick to avoid division in square root computation is to run Newton's method on the inverse square root instead. The iteration is then $R' = \frac{1}{2} R(3 - R^2 X^2)$ where R converges (quadratically) to $1/\sqrt{X^2}$. To get the square root, just multiply R by X^2. This leads to a reciprocal version of the Wilson-Burg algorithm. $A'/A + A'/A = 3 - AAS$ Here is how it can work: Construct an inverse autocorrelation — for example, an ideal isotropic smoother; make a guess for A (min-phase roughener); iterate: (1) compute $3 - AA * S$, (2) take its causal part, (3) convolve with A to get A'. Each iteration involves just three convolutions (could be even done without helix).

4.5. HELIX LOW-CUT FILTER

If you want to see some tracks on the side of a hill, you want to subtract the hill and see only the tracks. Usually, however, you don't have a very good model for the hill. As an expedient you could apply a low-cut filter to remove all slowly variable functions of altitude. In chapter 1 we found the Sea of Galilee in Figure 1.3 to be too smooth for viewing pleasure so we made the roughened versions in Figure 1.6 using a filter based on equation (1.26), a one-dimensional filter that we could apply over the x-axis or the y-axis. In Fourier space such a filter has a response function of k_x or a function of k_y. The isotropy of physical space tells us it would be more logical to design a filter that is a function of $k_x^2 + k_y^2$. In Figure 4.7 we saw that the helix derivative $\mathbf{H}$ does a nice job. The Fourier magnitude of its impulse response is $k_r = \sqrt{k_x^2 + k_y^2}$. There is a little anisotropy connected with phase (which way should we wind the helix, on x or y?) but it is not nearly so severe as that of either component of the gradient, the two components having wholly different spectra, amplitude $|k_x|$ or $|k_y|$.

It is nice having the 2-D helix derivative, but we can imagine even nicer 2-D low-cut filters. In one dimension, equation (1.26) and (2.13) we designed a filters

with an adjustable parameter, a cutoff frequency. We don't have such an object in 2-D so I set out to define one. It came out somewhat abstract and complicated, and didn't work very well, but along the way I found a simpler parameter that is very effective in practice. We'll look at it first.

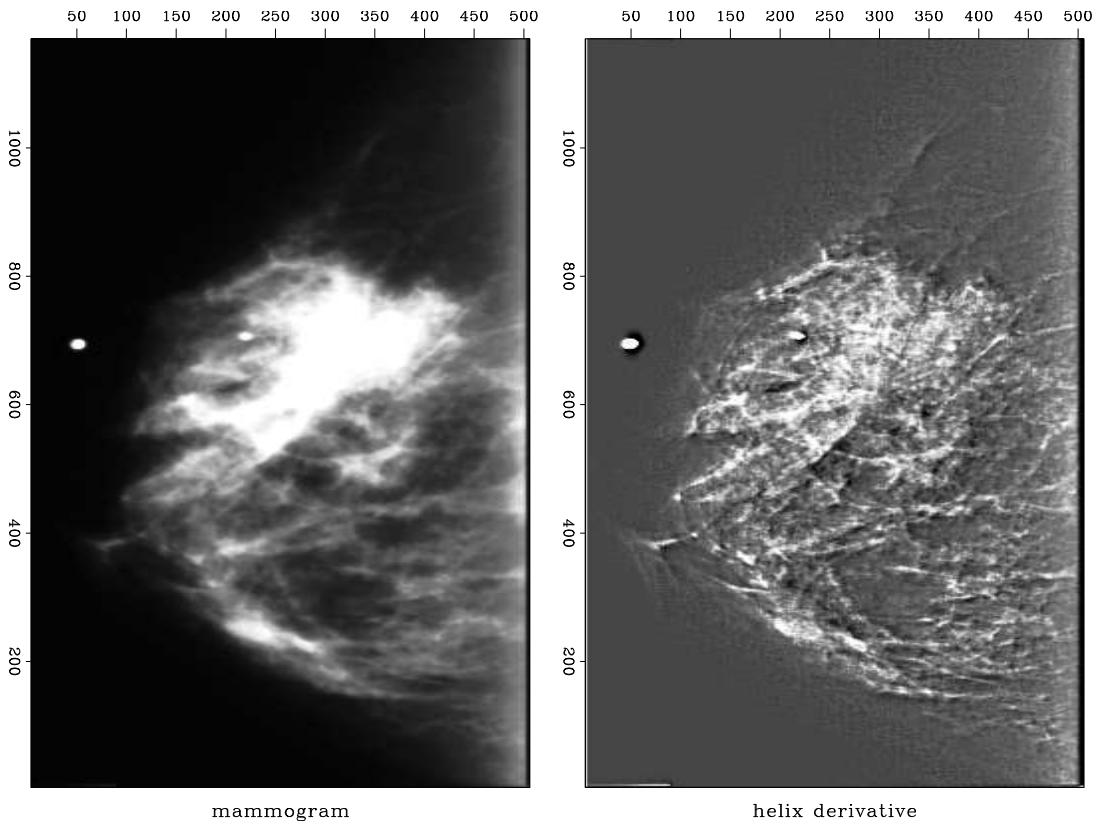


Figure 4.10: Mammogram (medical X-ray). The cancer is the "spoked wheel." (I apologize for the inability of paper publishing technology to exhibit a clear grey image.) The white circles are metal foil used for navigation. The little halo around a circle exhibits the impulse response of the helix derivative. hlx-mam [ER,M]

First I had a problem preparing Figure 4.10. It shows shows the application of the helix derivative to a medical X-ray. The problem was that the original X-ray was all positive values of brightness so there was a massive amount of spatial low frequency present. Obviously an x-derivative or a y-derivative would eliminate the low frequency, but the helix derivative did not. This unpleasant surprise arises because the filter in equation (4.11) was truncated after a finite number of terms. Adding up the terms actually displayed in equation (4.11), they sum to .183 whereas theoretically the sum of all the terms should be zero. From the ratio of .183/1.791 we can say that the filter pushes zero frequency amplitude 90% of the way to zero value. When the image contains very much zero frequency amplitude, this is not good enough. Better results could be obtained with more coefficients, and I did use more coefficients, but simply removing the mean saved me from needing a costly number of filter coefficients.

We can visualize a plot of the magnitude of the 2-D Fourier transform of the filter (4.11). It is a 2-D function of k_x and k_y and it should resemble $k_r = \sqrt{k_x^2 + k_y^2}$. It does look like this even when the filter (4.11) has been truncated. The point of the cone $k_r = \sqrt{k_x^2 + k_y^2}$ becomes rounded and the truncated approximation of k_r

does not reach zero at the origin of the (k_x, k_y)-plane. We can force it to vanish at zero frequency by subtracting .183 from the lead coefficient 1.791. I did not do that subtraction in Figure 4.11 which explains the whiteness in the middle of the lake.

Now let us return to my more logical but less effective approach. I prepared a half dozen medical X-rays like Figure 4.10. The doctor brought her young son to my office one evening to evaluate the results. In a dark room I would show the original X-ray on a big screen and then suddenly switch to the helix derivative. Every time I did this, her son would exclaim "Wow!" The doctor was not so easily impressed, however. She was not accustomed to the unfamiliar image. Fundamentally, the helix derivative applied to her data does compress the dynamic range making weaker features more readily discernable. We were sure of this from theory and from various geophysical examples. The subjective problem was her unfamiliarity with our display. I found that I could always spot anomalies more quickly on the filtered display, but then I would feel more comfortable when I would discover those same anomalies also present (though less evident) in the original data. Thinking this through, I decided the doctor would likely have been more impressed had I used a spatial lowcut filter instead of the helix derivative. That would have left the details of her image (above the cutoff frequency) unchanged altering only the low

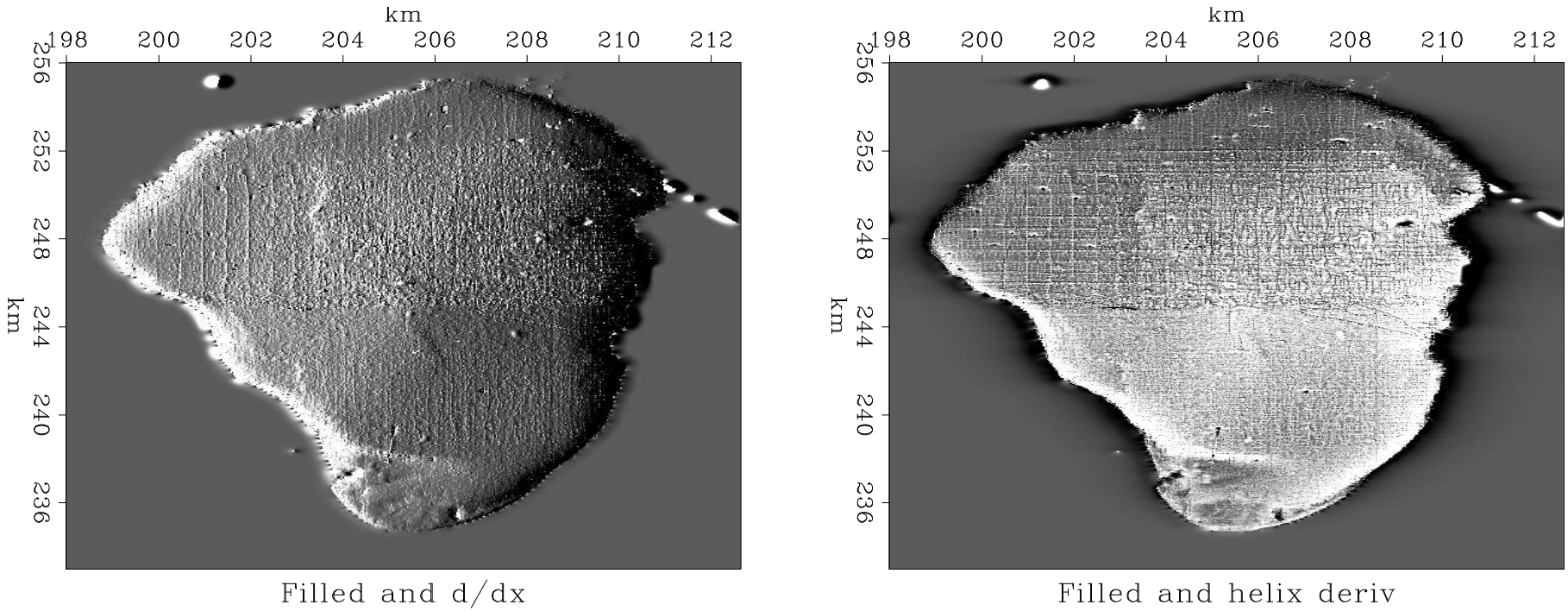


Figure 4.11: Galilee roughened by gradient and by helical derivative. hlx-helgal [ER,M]

frequencies, thereby allowing me to increase the gain.

In 1-D we easily make a low-cut filter by compounding a first derivative (which destroys low frequencies) with a leaky integration (which undoes the derivative at all other frequencies). We can do likewise with a second derivative. In Z-transform notation, we would use something like $(-Z^{-1}+2.00-Z)/(-Z^{-1}+2.01-Z)$. (The numerical choice of the .01 controls the cutoff frequency.) We could use spectral factorization to break this spectrum into causal and anticausal factors. The analogous filter in 2-D is $-\nabla^2/(-\nabla^2+k_0^2)$ which could also be factored as we did the helix derivative. I tried it. I ran into the problem that my helix derivative operator had a practical built-in parameter, the number of coefficients, which also behaves like a cutoff frequency. If I were to continue this project, I would use expressions for $-\nabla^2/(-\nabla^2+k_0^2)$ directly in the Fourier domain where there is only one adjustable parameter, the cutoff frequency k_0, and there is no filter length to confuse the issue and puff-up the costs.

A final word about the doctor. As she was about to leave my office she suddenly asked whether I had scratched one of her X-rays. We were looking at the helix derivative and it did seem to show a big scratch. What should have been a line was broken into a string of dots. I apologized in advance and handed her the original

film negatives which she proceeded to inspect. "Oh," she said, "Bad news. There are calcification nodules along the ducts." So the scratch was not a scratch, but an important detail that had not been noticed on the original X-ray.

In preparing an illustration for here, I learned one more lesson. The scratch was small, so I enlarged a small portion of the mammogram for display. The very process of selecting a small portion followed by scaling the amplitude between maximum and minimum darkness of printer ink had the effect enhancing the visibility of the scratch on the mammogram itself. Now Figure 4.12 shows it to be perhaps even clearer than on the helix derivative.

An operator for applying the helix filter is `helderiv` /prog:helderiv. helderiv

4.6. THE MULTIDIMENSIONAL HELIX

Till now the helix idea was discussed as if it were merely a two-dimensional concept. Here we explore its multidimensional nature. Our main goal is to do multidimensional convolution with a one-dimensional convolution program. This allows us to do multidimensional deconvolution with a one-dimensional deconvolutional

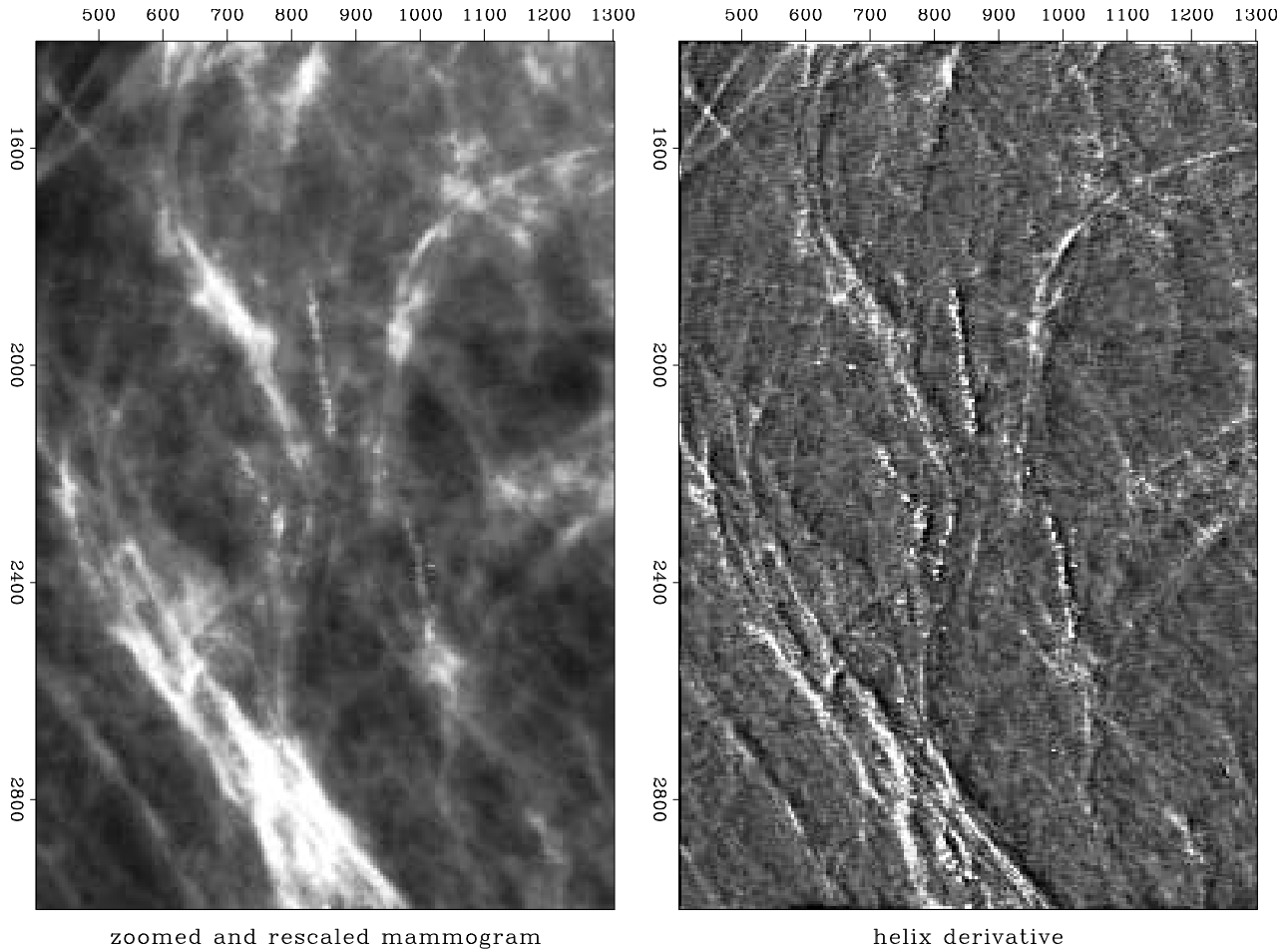


Figure 4.12: Not a scratch hlx-scratch [ER,M]

```
module helderiv {
use lapfac
use helicon
type( filter), private :: aa
#%  _init( n1, na, eps)
integer, intent (in) :: n1, na
real,    intent (in) :: eps
aa = lapfac2( eps, n1, na)
call helicon_init( aa)
#% _lop (pp, qq)
integer stat1
stat1 = helicon_lop( adj, .false.,  pp, qq)
#% _close
call deallocatehelix( aa)
}
```

Back

program which is “magic”, i.e. many novel applications will follow.

We do multidimensional deconvolution with causal (one-sided) one-dimensional filters. Equation (4.7) shows such a one-sided filter as it appears at the end of a 2-D helix. Figure 4.13 shows it in three dimensions. The top plane in Figure 4.13 is the 2-D filter seen in equation (4.7). The top plane can be visualized as the area around the end of a helix. Above the top plane are zero-valued anticausal filter coefficients.

It is natural to ask, “why not put the ‘1’ on a corner of the cube?” We could do that, but that is not the most general possible form. A special case of Figure 4.13, stuffing much of the volume with lots of zeros would amount to a ‘1’ on a corner. On the other hand, if we assert the basic form has a ‘1’ on a corner we cannot get Figure 4.13 as a special case. In a later chapter we’ll see that we often need as many coefficients as we can have near the ‘1’. In Figure 4.13 we lose only those neighboring coefficients that 1-D causality requires.

Geometrically, the three-dimensional generalization of a helix is like string on a spool, but that analogy does not illuminate our underlying conspiracy, which is to represent multidimensional convolution and deconvolution as one-dimensional.

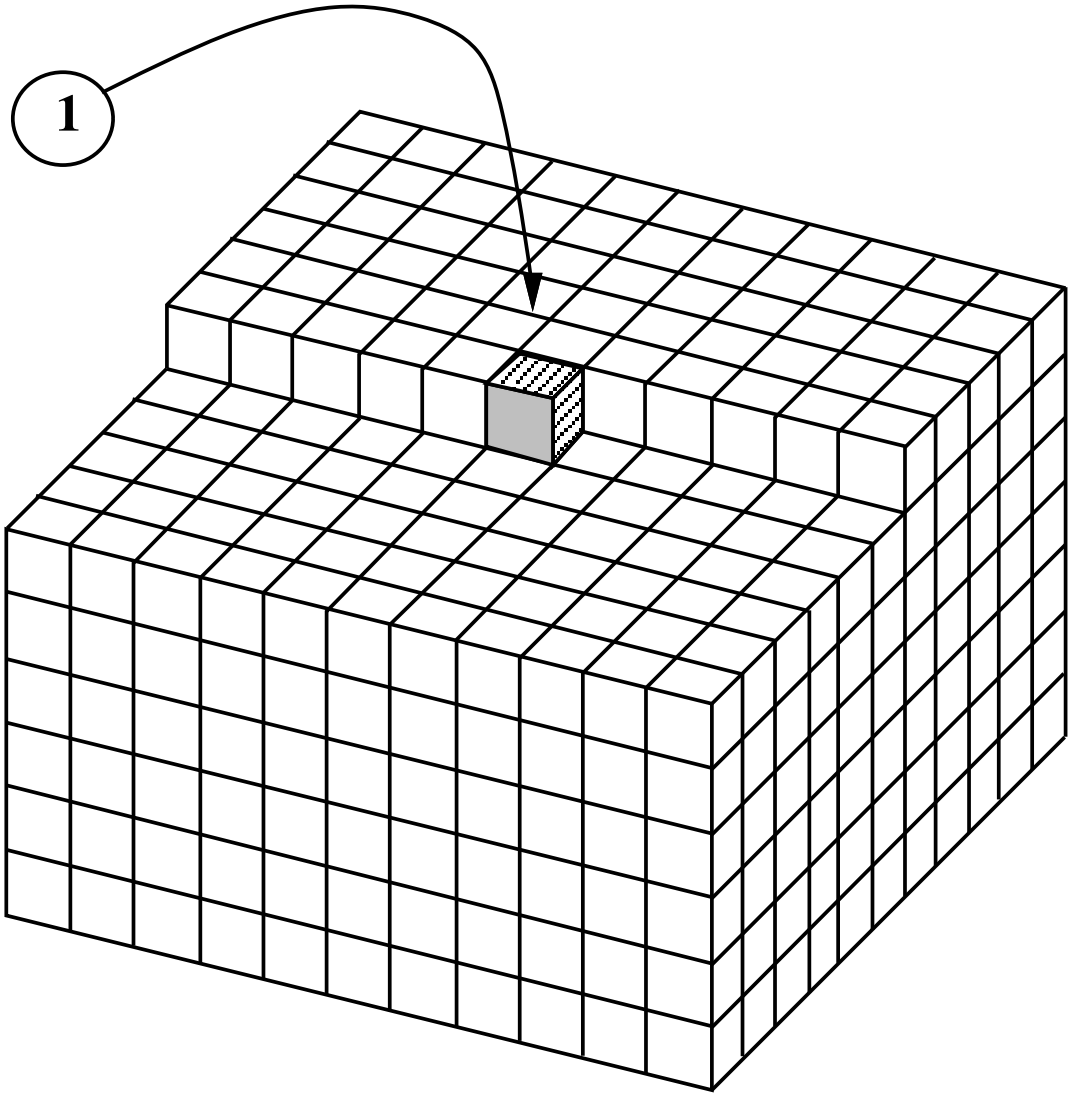


Figure 4.13: A 3-D causal filter at the starting end of a 3-D helix. hlx-3dpef [NR]

4.7. SUBSCRIPTING A MULTIDIMENSIONAL HELIX

Basic utilities transform back and forth between multidimensional matrix coordinates and helix coordinates. The essential module used repeatedly in applications later in this book is `createhelixmod` /prog:createhelixmod. We begin here from its intricate underpinnings.

Fortran77 has a concept of a multidimensional array being equivalent to a one-dimensional array. Given that the hypercube specification `nd=(n1,n2,n3,...)` defines the storage `dimension` of a data array, we can refer to a data element as either `dd(i1,i2,i3,...)` or `dd( i1 +n1*(i2-1) +n1*n2*(i3-1) +...)`. The helix says to refer to the multidimensional data by its equivalent one-dimensional index (sometimes called its vector subscript or linear subscript).

The filter, however, is a much more complicated story than the data: First, we require all filters to be causal. In other words, the Laplacian doesn't fit very well, since it is intrinsically noncausal. If you really want noncausal filters, you will need to provide your own time shifts outside the tools supplied here. Second, a filter is usually a small hypercube, say `aa(a1,a2,a3,...)` and would often be stored as

such. For the helix we must store it in a special one-dimensional form. Either way, the numbers `na= (a1,a2,a3,...)` specify the dimension of the hypercube. In cube form, the entire cube could be indexed multidimensionally as `aa(i1,i2,...)` or it could be indexed one-dimensionally as `aa(ia,1,1,...)` or sometimes[3] `aa(ia)` by letting `ia` cover a large range. When a filter cube is stored in its normal "tightly packed" form the formula for computing its one-dimensional index `ia` is

```
ia = i1 +a1*(i2-1) +a1*a2*(i3-1) + ...
```

When the filter cube is stored in an array with the same dimensions as the data, `data(n1,n2,n3,...)`, the formula for `ia` is

```
ia = i1 +n1*(i2-1) +n1*n2*(i3-1) + ...
```

---

[3] Some programming minutia: Fortran77 does not allow you to refer to an array by both its cartesian coordinates and by its linear subscript in the same subroutine. To access it both ways, you need a subroutine call, or you dimension it as `data(n1,n2,...)` and then you refer to it as `data(id,1,1,...)`. Fortran90 follows the same rule outside modules. Where modules use other modules, the compiler does not allow you to refer to data both ways, unless the array is declared as `allocatable`.

```
module cartesian {       # index transform (vector to matrix) and its inverse
contains
  subroutine line2cart( nn, i, ii) {
    integer, dimension( :), intent( in) :: nn    # cartesian axes (n1,n2,n3,...)
    integer, dimension( :), intent(out) :: ii    # cartesn coords (i1,i2,i3,...)
    integer                , intent( in) :: i     # equivalent 1-D linear index
    integer                              :: axis, n123
    n123 = 1
    do axis = 1, size(  nn) {
        ii( axis) = mod( ( i-1)/n123, nn( axis)) + 1
        n123 = n123 * nn( axis)
        }
    }
  subroutine cart2line( nn, ii, i) {
    integer, dimension( :), intent( in) :: nn, ii
    integer                              :: i, axis, n123
    n123 = 1;    i = 1
    do axis = 1, size(  nn) {
        i = i + ( ii( axis)-1)*n123
        n123 = n123 * nn( axis)
        }
    }
}
```

Back

The fortran compiler knows how to convert from the multidimensional cartesian indices to the linear index. We will need to do that, as well as the converse. Module `cartesian` below contains two subroutines that explicitly provide us the transformations between the linear index `i` and the multidimensional indices `ii= (i1,i2,...)`. The two subroutines have the logical names `cart2line` and `line2cart`. cartesian

The fortran linear index is closely related to the helix. There is one major difference, however, and that is the origin of the coordinates. To convert from the linear index to the helix lag coordinate, we need to subtract the fortran linear index of the "1.0" which is usually taken at `center= (1+a1/2, 1+a2/2, ..., 1)`. (On the last dimension, there is no shift because nobody stores the volume of zero values that would occur before the 1.0.) The `cartesian` module fails for negative subscripts. Thus we need to be careful to avoid thinking of the filter's 1.0 (shown in Figure 4.13) as the origin of the multidimensional coordinate system although the 1.0 is the origin in the one-dimensional coordinate system.

Even in one dimension (see the matrix in equation (1.4)), to define a filter *operator* we need to know not only filter coefficients and a filter length, but we also need to know the data length. To define a multidimensional filter using the helix idea,

besides the properties intrinsic to the filter, we also need to know the circumference of the helix, i.e., the length on the 1-axis of the data's hypercube as well as the other dimensions `nd=(n1,n2,...)` of the data's hypecube.

Thinking about convolution on the helix, it is natural to think about the filter and data being stored in the same way, that is, by reference to the data size. This would waste so much space, however, that our helix filter module `helix` /prog:helix instead stores the filter coefficients in one vector and their lags in another. The `i`-th coefficient value of the filter goes in `aa%flt(i)` and the `i`-th lag `ia(i)` goes in `aa%lag(i)`. The lags are the same as the fortran linear index except for the overall shift of the 1.0 of a cube of data dimension `nd`. Our module for convolution on a helix, `helicon` /prog:helicon, has already an implicit "1.0" at the filter's zero lag so we do not store it. (It is an error to do so.)

Module `createhelixmod` /prog:createhelixmod allocates memory for a helix filter and builds filter lags along the helix from the hypercube description. The hypercube description is not the literal cube seen in Figure 4.13 but some integers specifying that cube: the data cube dimensions `nd`, likewise the filter cube dimensions `na`, the parameter `center` identifying the location of the filter's "1.0", and a `gap` parameter used in a later chapter. To find the lag table, module `createhelix-`

```
module createhelixmod {                          # Create helix filter lags and mis
use helix
use cartesian
contains
  function createhelix( nd, center, gap, na)  result( aa) {
    type( filter)                         :: aa       # needed by helicon.
    integer, dimension(:), intent(in) :: nd, na   # data and filter axes
    integer, dimension(:), intent(in) :: center   # normally (na1/2,na2/2,...,1)
    integer, dimension(:), intent(in) :: gap      # normally ( 0,   0,  0,...,0)
    integer, dimension( size( nd))    :: ii       # cartesian indexes
    integer                               :: na123, ia, ndim, nh, lag0a,lag0d
    integer, dimension(:), allocatable:: lag
           nh= 0;   na123 = product( na);   ndim = size( nd)
    allocate(  lag( na123 ) )                 # filter cube size
    call cart2line ( na, center, lag0a)   # lag0a = index pointing to the "1.0"
    do ia = 1+lag0a, na123 {                  #    ia  is fortran linear index.
       call line2cart( na, ia, ii)            # ii(ia) is fortran array  indices.
       if( any( ii <= gap))    next           # ignore some locations
       nh = nh + 1                            # got another live one.
       call cart2line( nd, ii, lag(nh))   # get its fortran linear index
       }
    call cart2line(  nd, center, lag0d)   # lag0d is center shift for nd_cube
    call allocatehelix( aa, nh)               # nh becomes size of filter on helix.
    aa%lag = lag(1:nh) - lag0d;               # lag = fortran_linear_index - center
    aa%flt = 0.0;              deallocate( lag)
    }
}
```

Back

`mod` first finds the fortran linear index of the `center` point on the filter hypercube. Everything before that has negative lag on the helix and can be ignored. (Likewise, in a later chapter we see a `gap` parameter that effectively sets even more filter coefficients to zero so their lags can be ignored too.) Then it sweeps from the center point over the rest of the filter hypercube calculating for a data-sized cube `nd`, the fortran linear index of each filter element. createhelixmod Near the end of the code you see the calculation of a parameter `lag0d`. This is the count of the number of zeros that a data-sized fortran array would store in a filter cube before the filter's 1.0. We need to subtract this shift from the filter's fortran linear index to get the lag on the helix.

A filter can be represented literally as a multidimensional cube like equation (4.7) shows us in two dimensions or like Figure 4.13 shows us in three dimensions. Unlike the helical form, in literal cube form, the zeros preceding the "1.0" are explicitly present so `lag0` needs to be added back in to get the fortran subscript. To convert a helix filter `aa` to fortran's multidimensional hypercube `cube(n1,n2,...)` is module `box`: box The `box` module is normally used to display or manipulate a filter that was estimated in helical form (usually estimated by the least-squares method).

```
module box {         # Convert helix filter to hypercube: cube(na(1),na(2),...)
use helix
use cartesian
contains
  subroutine boxn( nd, center, na, aa, cube) {
    integer, dimension (:), intent( in)  :: nd, center, na    # (ndim)
    type( filter),          intent( in)  :: aa
    real,    dimension( :), intent( out) :: cube
    integer, dimension( size( nd))       :: ii
    integer                              :: j, lag0a, lag0d, id, ia
    cube = 0.;                          # cube=0
    call cart2line( na, center, lag0a)  # locate the 1.0 in the na_cube.
    cube( lag0a) = 1.                   # place it.
    call cart2line( nd, center, lag0d)  # locate the 1.0 in the nd_cube.
    do j = 1, size( aa%lag) {           # inspect the entire helix
       id = aa%lag( j) + lag0d          # index = helix_lag + center_d
       call line2cart( nd, id, ii)      # ii(id) = cartesian indices
       call cart2line( na,     ii, ia)  # ia(ii) = linear index in aa
       cube( ia) = aa%flt( j)           # copy the filter coefficient
       }
    }
}
```

Back

```
module unbox {                              # helixfilter aa = cube(a1,a2,...)
use helix
use cartesian
contains
  function unboxn( nd, center, na, cube)  result( aa) {
    type( filter)                          :: aa
    integer, dimension( :), intent( in) :: nd, center, na      # (ndim)
    real,    dimension( :), intent( in) :: cube                # cube(a1,a2,...)
    logical, dimension( size( cube))    :: keep                # keep(a1*a2*...)
    integer, dimension( size( nd))      :: ii                  # (ndim)
    integer                             :: ic, lag0a, lag0d, i, h
    call cart2line(  na, center, lag0a)
    call cart2line(  nd, center, lag0d)
    keep = ( abs( cube) > epsilon( cube))     # epsilon is a Fortran intrinsic
    keep( lag0a) = .false.                    # throw away the 1.0.
    call allocatehelix( aa, count( keep));  h = 0
    do ic = 1, size( cube) {                                  # sweep cube
           if( keep( ic) ) {                 h = h + 1  # only the keepers
                call line2cart(  na, ic, ii)           # ii(ic)= indices on na
                call cart2line(  nd,     ii, i)       # i     = index   on nd
                aa%lag( h) = i - lag0d                 # lag = index - center
                aa%flt( h) = cube( ic)                 # copy coefs.
                }
           }
    }
}
```

Back

The inverse process to `box` is to convert a fortran hypercube to a helix filter. For this we have module `unbox`. It abandons all zero-valued coefficients such as those that should be zero before the box's 1.0. It abandons the "1.0" as well, because it is implicitly present in the helix convolution module `helicon` /prog:helicon. unbox

An example of using `unbox` would be copying some numbers such as the factored laplacian in equation (4.11) into a cube and then converting it to a helix.

A reasonable arrangement for a small 3-D filter is `na=(5,3,2)` and `center=(3,2,1)`. Using these arguments, I used `createhelixmod` /prog:createhelixmod to create a filter. I set all the helix filter coefficients to 2. Then I used module `box` /prog:box to put it in a convenient form for display. After this conversion, the coefficient `aa(3,2,1)` is 1, not 2. Finally, I printed it:

```
0.000  0.000  0.000  0.000  0.000
0.000  0.000  1.000  2.000  2.000
2.000  2.000  2.000  2.000  2.000
-------------------------------
2.000  2.000  2.000  2.000  2.000
2.000  2.000  2.000  2.000  2.000
2.000  2.000  2.000  2.000  2.000
```

Different data sets have different sizes. To convert a helix filter from one data size to another, we could drop the filter into a cube with module `cube`. Then we could extract it with module `unbox` specifying any data set size we wish. Instead we use module `regrid` prepared by Sergey Fomel which does the job without reference to an underlying filter cube. He explains his `regrid` module thus:

> Imagine a filter being cut out of a piece of paper and glued on another paper, which is then rolled to form a helix.
>
> We start by picking a random point (let's call it `rand`) in the cartesian grid and placing the filter so that its center (the leading 1.0) is on top of that point. `rand` should be larger than (or equal to) `center` and smaller than `min (nold, nnew)`, otherwise the filter might stick outside the grid (our piece of paper.) `rand=nold/2` will do (assuming the filter is small), although nothing should change if you replace `nold/2` with a random integer array between `center` and `nold - na`.
>
> The linear coordinate of `rand` is `h0` on the old helix and `h1` on the new helix. Recall that the helix lags `aa%lag` are relative to the center. Therefore, we need to add `h0` to get the absolute helix coordinate (`h`). Likewise, we need to subtract `h1` to return to a relative coordinate

system.

regrid

```
module regrid {         # convert a helix filter from one size data to another
use helix
use cartesian
contains
  subroutine regridn( nold, nnew, aa) {
    integer, dimension (:), intent (in) :: nold, nnew  # old and new helix grid
    type( filter)                       :: aa
    integer, dimension( size( nold))    :: ii
    integer                             :: i, h0, h1, h
    call cart2line( nold, nold/2, h0)   # lag of any near middle point on nold
    call cart2line( nnew, nold/2, h1)   # lag                           on nnew
    do i = 1, size( aa%lag) {           # forall given filter coefficients
       h = aa%lag( i) + h0              # what is this?
       call line2cart( nold, h, ii)     #
       call cart2line( nnew,    ii, h)  #
       aa%lag( i) = h - h1              # what is this
    }
  }
}
```

Back

Chapter 5

Preconditioning

When I first realized that practical imaging methods in widespread industrial use amounted merely to the adjoint of forward modeling, I (and others) thought an easy way to achieve fame and fortune would be to introduce the first steps towards in-

version along the lines of Chapter 2. Although inversion generally requires a prohibitive number of steps, I felt that moving in the gradient direction, the direction of steepest descent, would move us rapidly in the direction of practical improvements. This turned out to be optimistic. It was too slow. But then I learned about the conjugate gradient method that spectacularly overcomes a well-known speed problem with the method of steepest descents. I came to realize that it was still too slow. I learned this by watching the convergence in Figure 5.6. This led me to the helix method in Chapter 4. Here we'll see how it speeds many applications.

We'll also come to understand why the gradient is such a poor direction both for steepest descent and for conjugate gradients. An indication of our path is found in the contrast between and exact solution $\mathbf{m} = (\mathbf{A}'\mathbf{A})^{-1}\mathbf{A}'\mathbf{d}$ and the gradient $\Delta\mathbf{m} = \mathbf{A}'\mathbf{d}$ (which is the first step starting from $\mathbf{m} = \mathbf{0}$). Notice that $\Delta\mathbf{m}$ differs from $\mathbf{m}$ by the factor $(\mathbf{A}'\mathbf{A})^{-1}$. This factor is sometimes called a spectrum and in some situations it literally is a frequency spectrum. In these cases, $\Delta\mathbf{m}$ simply gets a different spectrum from $\mathbf{m}$ and many iterations are required to fix it. Here we'll find that for many problems, "preconditioning" with the helix is a better way.

5.1. PRECONDITIONED DATA FITTING

Iterative methods (like conjugate-directions) can sometimes be accelerated by a change of variables. The simplest change of variable is called a "trial solution". Formally, we write the solution as

$$\mathbf{m} \quad = \quad \mathbf{Sp} \tag{5.1}$$

where $\mathbf{m}$ is the map we seek, columns of the matrix $\mathbf{S}$ are "shapes" that we like, and coefficients in $\mathbf{p}$ are unknown coefficients to select amounts of the favored shapes. The variables $\mathbf{p}$ are often called the "preconditioned variables". It is not necessary that $\mathbf{S}$ be an invertible matrix, but we'll see later that invertibility is helpful. Take this trial solution and insert it into a typical fitting goal

$$\mathbf{0} \quad \approx \quad \mathbf{Fm} - \mathbf{d} \tag{5.2}$$

and get

$$\mathbf{0} \quad \approx \quad \mathbf{FSp} - \mathbf{d} \tag{5.3}$$

We pass the operator $\mathbf{FS}$ to our iterative solver. After finding the best fitting $\mathbf{p}$, we merely evaluate $\mathbf{m} = \mathbf{Sp}$ to get the solution to the original problem.

We hope this change of variables has saved effort. For each iteration, there is a little more work: Instead of the iterative application of $\mathbf{F}$ and $\mathbf{F}'$ we have iterative application of $\mathbf{FS}$ and $\mathbf{S}'\mathbf{F}'$. Our hope is that the number of iterations decreases because we are clever, or because we have been lucky in our choice of $\mathbf{S}$. Hopefully, the extra work of the preconditioner operator $\mathbf{S}$ is not large compared to $\mathbf{F}$. If we should be so lucky that $\mathbf{S} = \mathbf{F}^{-1}$, then we get the solution immediately. Obviously we would try any guess with $\mathbf{S} \approx \mathbf{F}^{-1}$. Where I have known such $\mathbf{S}$ matrices, I have often found that convergence is accelerated, but not by much. Sometimes it is worth using $\mathbf{FS}$ for a while in the beginning, but later it is cheaper and faster to use only $\mathbf{F}$. A practitioner might regard the guess of $\mathbf{S}$ as prior information, like the guess of the initial model $\mathbf{m}_0$.

For a square matrix $\mathbf{S}$, the use of a preconditioner should not change the ultimate solution. Taking $\mathbf{S}$ to be a tall rectangular matrix, reduces the number of adjustable parameters, changes the solution, gets it quicker, but lower resolution.

5.1.1. Preconditioner with a starting guess

In many applications, for many reasons, we have a starting guess $\mathbf{m}_0$ of the solution. You might worry that you could not find the starting preconditioned variable $\mathbf{p}_0 = \mathbf{S}^{-1}\mathbf{m}_0$ because you did not know the inverse of $\mathbf{S}$. The way to avoid this problem is to reformulate the problem in terms of a new variable $\tilde{\mathbf{m}}$ where $\mathbf{m} = \tilde{\mathbf{m}} + \mathbf{m}_0$. Then $\mathbf{0} \approx \mathbf{F}\mathbf{m} - \mathbf{d}$ becomes $\mathbf{0} \approx \mathbf{F}\tilde{\mathbf{m}} - (\mathbf{d} - \mathbf{F}\mathbf{m}_0)$ or $\mathbf{0} \approx \mathbf{F}\tilde{\mathbf{m}} - \tilde{\mathbf{d}}$. Thus we have accomplished the goal of taking a problem with a nonzero starting model and converting it a problem of the same type with a zero starting model. Thus we do not need the inverse of $\mathbf{S}$ because the iteration starts from $\tilde{\mathbf{m}} = \mathbf{0}$ so $\mathbf{p}_0 = \mathbf{0}$.

5.2. PRECONDITIONING THE REGULARIZATION

The basic formulation of a geophysical estimation problem consists of setting up *two* goals, one for data fitting, and the other for model shaping. With two goals,

preconditioning is somewhat different. The two goals may be written as:

$$\mathbf{0} \quad \approx \quad \mathbf{Fm - d} \tag{5.4}$$

$$\mathbf{0} \quad \approx \quad \mathbf{Am} \tag{5.5}$$

which defines two residuals, a so-called “data residual” and a “model residual” that are usually minimized by conjugate-gradient, least-squares methods.

To fix ideas, let us examine a toy example. The data and the first three rows of the matrix below are random numbers truncated to integers. The model roughening operator **A** is a first differencing operator times 100.

d(m)	F(m,n)										iter	Norm
41.	-55.	-90.	-24.	-13.	-73.	61.	-27.	-19.	23.	-55.	1	20.003965
33.	8.	-86.	72.	87.	-41.	-3.	-29.	29.	-66.	50.	2	12.147801
-58.	84.	-49.	80.	44.	-52.	-51.	8.	86.	77.	50.	3	8.943936
0.	100.	0.	0.	0.	0.	0.	0.	0.	0.	0.	4	6.045171
0.	-100.	100.	0.	0.	0.	0.	0.	0.	0.	0.	5	2.647375
0.	0.	-100.	100.	0.	0.	0.	0.	0.	0.	0.	6	0.792384
0.	0.	0.	-100.	100.	0.	0.	0.	0.	0.	0.	7	0.460833
0.	0.	0.	0.	-100.	100.	0.	0.	0.	0.	0.	8	0.083012
0.	0.	0.	0.	0.	-100.	100.	0.	0.	0.	0.	9	0.005420
0.	0.	0.	0.	0.	0.	-100.	100.	0.	0.	0.	10	0.000005
0.	0.	0.	0.	0.	0.	0.	-100.	100.	0.	0.	11	0.000000
0.	0.	0.	0.	0.	0.	0.	0.	-100.	100.	0.	12	0.000000
0.	0.	0.	0.	0.	0.	0.	0.	0.	-100.	100.	13	0.000000

Notice at the tenth iteration, the residual suddenly plunges 4 significant digits. Since there are ten unknowns and the matrix is obviously full-rank, conjugate-

gradient theory tells us to expect the exact solution at the tenth iteration. This is the first miracle of conjugate gradients. (The residual actually does not drop to zero. What is printed in the `Norm` column is the square root of the sum of the squares of the residual components at the `iter`-th iteration minus that at the last interation.)

5.2.1. The second miracle of conjugate gradients

The second miracle of conjugate gradients is exhibited below. The data and data fitting matrix are the same, but the model damping is simplified.

```
d(m)    F(m,n)                                                  iter  Norm
---    ------------------------------------------------------   ----  ------
----
 41.    -55. -90. -24. -13. -73.  61. -27. -19.  23. -55.      1  3.6441068
 33.      8. -86.  72.  87. -41.  -3. -29.  29. -66.  50.      2  0.3126989
-58.     84. -49.  80.  44. -52. -51.   8.  86.  77.  50.      3 -0.0000002
  0.    100.   0.   0.   0.   0.   0.   0.   0.   0.   0.      4 -0.0000006
  0.      0. 100.   0.   0.   0.   0.   0.   0.   0.   0.      5 -0.0000008
  0.      0.   0. 100.   0.   0.   0.   0.   0.   0.   0.      6 -0.0000006
  0.      0.   0.   0. 100.   0.   0.   0.   0.   0.   0.      7 -0.0000008
  0.      0.   0.   0.   0. 100.   0.   0.   0.   0.   0.      8 -0.0000007
  0.      0.   0.   0.   0.   0. 100.   0.   0.   0.   0.      9 -0.0000003
  0.      0.   0.   0.   0.   0.   0. 100.   0.   0.   0.     10 -0.0000003
  0.      0.   0.   0.   0.   0.   0.   0. 100.   0.   0.     11 -0.0000001
  0.      0.   0.   0.   0.   0.   0.   0.   0. 100.   0.     12  0.0000000
  0.      0.   0.   0.   0.   0.   0.   0.   0.   0. 100.     13  0.0000000
```

Even though the matrix is full-rank, we see the residual drop about 6 decimal places after the third iteration! This convergence behavior is well known in the computa-

tional mathematics literature. Despite its practical importance, it doesn't seem to have a name or identified discoverer. So I call it the "second miracle."

Practitioners usually don't like the identity operator for model-shaping. Generally they prefer to penalize wiggliness. For practitioners, the lesson of the second miracle of conjugate gradients is that we have a choice of many iterations, or learning to transform independent variables so that the regularization operator becomes an identity matrix. Basically, such a transformation reduces the iteration count from something about the size of the model space to something about the size of the data space. Such a transformation is called preconditioning. In practice, data is often accumulated in bins. Then the iteration count is reduced (in principle) to the count of full bins and should be independent of the count of the empty bins. This allows refining the bins, enhancing the resolution.

More generally, the model goal $\mathbf{0} \approx \mathbf{Am}$ introduces a roughening operator like a gradient, Laplacian (and in chapter 6 a Prediction-Error Filter (PEF)). Thus the model goal is usually a filter, unlike the data-fitting goal which involves all manner of geometry and physics. When the model goal is a filter its inverse is also a filter. Of course this includes multidimensional filters with a helix.

The preconditioning transformation $\mathbf{m} = \mathbf{Sp}$ gives us

$$\begin{aligned} \mathbf{0} &\approx \mathbf{FSp} - \mathbf{d} \\ \mathbf{0} &\approx \mathbf{ASp} \end{aligned} \tag{5.6}$$

The operator $\mathbf{A}$ is a roughener while $\mathbf{S}$ is a smoother. The choices of both $\mathbf{A}$ and $\mathbf{S}$ are somewhat subjective. This suggests that we eliminate $\mathbf{A}$ altogether by *defining* it to be proportional to the inverse of $\mathbf{S}$, thus $\mathbf{AS} = \mathbf{I}$. The fitting goals become

$$\begin{aligned} \mathbf{0} &\approx \mathbf{FSp} - \mathbf{d} \\ \mathbf{0} &\approx \epsilon\, \mathbf{p} \end{aligned} \tag{5.7}$$

which enables us to benefit from the "second miracle". After finding $\mathbf{p}$, we obtain the final model with $\mathbf{m} = \mathbf{Sp}$.

5.2.2. Importance of scaling

Another simple toy example shows us the importance of scaling. We use the same example as above except that the i-th column is multiplied by $i/10$ which means the i-th model variable has been divided by $i/10$.

d(m)	F(m,n)										iter	Norm
41.	-6.	-18.	-7.	-5.	-36.	37.	-19.	-15.	21.	-55.	1	11.5954484
33.	1.	-17.	22.	35.	-20.	-2.	-20.	23.	-59.	50.	2	6.9733777
-58.	8.	-10.	24.	18.	-26.	-31.	6.	69.	69.	50.	3	5.6441440
0.	10.	0.	0.	0.	0.	0.	0.	0.	0.	0.	4	4.3211817
0.	0.	20.	0.	0.	0.	0.	0.	0.	0.	0.	5	2.6475520
0.	0.	0.	30.	0.	0.	0.	0.	0.	0.	0.	6	2.0163135
0.	0.	0.	0.	40.	0.	0.	0.	0.	0.	0.	7	1.2321997
0.	0.	0.	0.	0.	50.	0.	0.	0.	0.	0.	8	0.3664920
0.	0.	0.	0.	0.	0.	60.	0.	0.	0.	0.	9	0.2852894
0.	0.	0.	0.	0.	0.	0.	70.	0.	0.	0.	10	0.0671241
0.	0.	0.	0.	0.	0.	0.	0.	80.	0.	0.	11	0.0037428
0.	0.	0.	0.	0.	0.	0.	0.	0.	90.	0.	12	-0.0000004
0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	100.	13	0.0000000

We observe that solving the same problem for the scaled variables has required a severe increase in the number of iterations required to get the solution. We lost the

benefit of the second CG miracle. Even the rapid convergence predicted for the 10-th iteration is delayed until the 12-th.

5.2.3. Statistical interpretation

This book is not a statistics book. Never-the-less, many of you have some statistical knowledge that allows you a statistical interpretation of these views of preconditioning.

A statistical concept is that we can combine many streams of random numbers into a composite model. Each stream of random numbers is generally taken to be uncorrelated with the others, to have zero mean, and to have the same variance as all the others. This is often abbreviated as IID, denoting Independent, Identically Distributed. Linear combinations like filtering and weighting operations of these IID random streams can build correlated random functions much like those observed in geophysics. A geophysical practitioner seeks to do the inverse, to operate on the correlated unequal random variables and create the statistical ideal random streams. The identity matrix required for the "second miracle", and our search for a good preconditioning transformation are related ideas. The relationship will become more

clear in chapter 6 when we learn how to estimate the best roughening operator $\mathbf{A}$ as a prediction-error filter.

Two philosophies to find a preconditioner:

1. Dream up a smoothing operator $\mathbf{S}$.
2. Estimate a prediction-error filter $\mathbf{A}$, and then use its inverse $\mathbf{S} = \mathbf{A}^{-1}$.

Deconvolution on a helix is an all-purpose preconditioning strategy for multi-dimensional model regularization.

The outstanding acceleration of convergence by preconditioning suggests that the philosophy of image creation by optimization has a dual orthonormality: First, Gauss (and common sense) tells us that the data residuals should be roughly equal in size. Likewise in Fourier space they should be roughly equal in size, which means they should be roughly white, i.e. orthonormal. (I use the word "orthonormal" because white means the autocorrelation is an impulse, which means the signal is statistically orthogonal to shifted versions of itself.) Second, to speed convergence

of iterative methods, we need a whiteness, another orthonormality, in the solution. The map image, the physical function that we seek, might not be itself white, so we should solve first for another variable, the whitened map image, and as a final step, transform it to the "natural colored" map.

5.2.4. The preconditioned solver

Summing up the ideas above, we start from fitting goals

$$\begin{aligned} \mathbf{0} &\approx \mathbf{Fm} - \mathbf{d} \\ \mathbf{0} &\approx \mathbf{Am} \end{aligned} \tag{5.8}$$

and we change variables from $\mathbf{m}$ to $\mathbf{p}$ using $\mathbf{m} = \mathbf{A}^{-1}\mathbf{p}$

$$\begin{aligned} \mathbf{0} &\approx \mathbf{Fm} - \mathbf{d} = \mathbf{FA}^{-1}\,\mathbf{p} - \mathbf{d} \\ \mathbf{0} &\approx \mathbf{Am} = \mathbf{I}\,\mathbf{p} \end{aligned} \tag{5.9}$$

Preconditioning means iteratively fitting by adjusting the $\mathbf{p}$ variables and then finding the model by using $\mathbf{m} = \mathbf{A}^{-1}\mathbf{p}$. A new reusable preconditioned solver is the module `solver_prc` /prog:solver_prc. Likewise the modeling operator $\mathbf{F}$ is called

```
module solver_prc_mod{                                  # 0 = W (F S J p - d)
  use chain0_mod + solver_report_mod                    # 0 =      I   p
  logical, parameter, private  :: T = .true., F = .false.
contains
  subroutine solver_prc( m,d, Fop, Sop, stepper, nSop, niter,eps &
  ,             Wop,Jop,p0,rm0,err,resd,resm,mmov,rmov,verb) {
    optional :: Wop,Jop,p0,rm0,err,resd,resm,mmov,rmov,verb
    interface { #-------------------------- begin definitions -----------
       integer function Fop(adj,add,m,d){real::m(:),d(:);logical,intent(in)::adj,add}
       integer function Sop(adj,add,m,d){real::m(:),d(:);logical,intent(in)::adj,add}
       integer function Wop(adj,add,m,d){real::m(:),d(:);logical,intent(in)::adj,add}
       integer function Jop(adj,add,m,d){real::m(:),d(:);logical,intent(in)::adj,add}
       integer function stepper(forget,m,g,rr,gg) {
         real, dimension(:)  ::        m,g,rr,gg
         logical             :: forget           }
    }
    real, dimension(:),    intent(in)          :: d, p0,rm0
    integer,               intent(in)          :: niter, nSop
    logical,               intent(in)          :: verb
    real,                  intent(in)          :: eps
    real, dimension(:),    intent(out)         :: m,err, resd,resm
    real, dimension(:,:),  intent(out)         ::        rmov,mmov
    real, dimension(size( m))                  :: p ,  g
    real, dimension(size( d) + nSop), target   :: rr, gg, tt
    real, dimension(:), pointer                :: rd, gd, td
    real, dimension(:), pointer                :: rm, gm, tm
    integer                                    :: iter, stat
    logical                                    :: forget
    rd => rr(1:size(d)); rm => rr(1+size(d):)
    gd => gg(1:size(d)); gm => gg(1+size(d):)
    td => tt(1:size(d)); tm => tt(1+size(d):)
    if(present( Wop)) stat=Wop(F,F,-d,rd) # begin initialization --------
    else rd = -d                                                  #Rd  = -W    d
    rm = 0.; if(present(rm0)) rm=rm0                              #Rm  =      Rm0
    if(present( p0)){ p=p0                                        # p  =       p0
       if(present( Wop)) call chain0(Wop,Fop,Sop,F,T,p,rd,tm,td)
       else              call chain0(    Fop,Sop,F,T,p,rd,tm   )#Rd +=  WFS  p0
       rm = rm + eps*p                                            #Rm +=   e I p0
```

`Fop` and the smoothing operator $\mathbf{A}^{-1}$ is called `Sop`. Details of the code are only slightly different from the regularized solver `solver_reg` /prog:solver_reg. You'll notice the code also allows a weighting function on the data residuals, allows for a starting $\mathbf{p}_0$, allows for masking constraints $\mathbf{J}$ on $\mathbf{p}$, and allows for scaling the regularization by an ϵ. solver_prc

5.2.5. Need for an invertible preconditioner

It is important to use regularization to solve many examples. It is important to precondition because in practice computer power is often a limiting factor. It is important to be able to begin from a nonzero starting solution because in nonlinear problems then we must restart from the result of an earlier solution. Putting all three requirements together leads to a little problem. It turns out the three together lead us to needing a preconditioning transformation that is invertible. Let us see why this is so.

$$\begin{aligned} \mathbf{0} &\approx \mathbf{Fm} - \mathbf{d} \\ \mathbf{0} &\approx \mathbf{Am} \end{aligned} \tag{5.10}$$

First we change variables from $\mathbf{m}$ to $\mathbf{u} = \mathbf{m} - \mathbf{m}_0$. Clearly $\mathbf{u}$ starts from $\mathbf{u}_0 = 0$, and $\mathbf{m} = \mathbf{u} + \mathbf{m}_0$. Then our regression pair becomes

$$\begin{aligned} \mathbf{0} &\approx \mathbf{F}\mathbf{u} + (\mathbf{F}\mathbf{m}_0 - \mathbf{d}) \\ \mathbf{0} &\approx \mathbf{A}\mathbf{u} + \mathbf{A}\mathbf{m}_0 \end{aligned} \tag{5.11}$$

This result differs from the original regression in only two minor ways, (1) revised data, and (2) a little more general form of the regularization, the extra term $\mathbf{A}\mathbf{m}_0$. Now let us introduce preconditioning. From the regularization we see this introduces the preconditioning variable $\mathbf{p} = \mathbf{A}\mathbf{u}$. Our regression pair becomes:

$$\begin{aligned} \mathbf{0} &\approx \mathbf{F}\mathbf{A}^{-1}\mathbf{p} + (\mathbf{F}\mathbf{m}_0 - \mathbf{d}) \\ \mathbf{0} &\approx \mathbf{p} + \mathbf{A}\mathbf{m}_0 \end{aligned} \tag{5.12}$$

Here is the problem: Now we require both $\mathbf{A}$ and $\mathbf{A}^{-1}$ operators. In 2- and 3-dimensional spaces we don't know very many operators with an easy inverse. Indeed, that is why I found myself pushed to come up with the helix methodology of the previous chapter – because it provides invertible operators for smoothing and roughening.

5.3. OPPORTUNITIES FOR SMART DIRECTIONS

Recall the fitting goals (5.13)

$$\begin{array}{lllllll} \mathbf{0} & \approx & \mathbf{r}_d & = & \mathbf{Fm} - \mathbf{d} & = & \mathbf{FA}^{-1}\ \mathbf{p} - \mathbf{d} \\ \mathbf{0} & \approx & \mathbf{r}_m & = & \mathbf{Am} & = & \mathbf{I}\ \ \mathbf{p} \end{array} \tag{5.13}$$

Without preconditioning we have the search direction

$$\Delta \mathbf{m}_{\text{bad}} \quad = \quad \begin{bmatrix} \mathbf{F}' & \mathbf{A}' \end{bmatrix} \begin{bmatrix} \mathbf{r}_d \\ \mathbf{r}_m \end{bmatrix} \tag{5.14}$$

and with preconditioning we have the search direction

$$\Delta \mathbf{p}_{\text{good}} \quad = \quad \begin{bmatrix} (\mathbf{FA}^{-1})' & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{r}_d \\ \mathbf{r}_m \end{bmatrix} \tag{5.15}$$

The essential feature of preconditioning is not that we perform the iterative optimization in terms of the variable $\mathbf{p}$. The essential feature is that we use a search direction that is a gradient with respect to $\mathbf{p}'$ not $\mathbf{m}'$. Using $\mathbf{Am} = \mathbf{p}$ we have $\mathbf{A}\Delta\mathbf{m} = \Delta\mathbf{p}$. This enables us to define a good search direction in model space.

$$\Delta \mathbf{m}_{\text{good}} \quad = \quad \mathbf{A}^{-1} \Delta \mathbf{p}_{\text{good}} \quad = \quad \mathbf{A}^{-1} (\mathbf{A}^{-1})' \mathbf{F}' \mathbf{r}_d + \mathbf{A}^{-1} \mathbf{r}_m \tag{5.16}$$

Define the gradient by $\mathbf{g} = \mathbf{F}'\mathbf{r}_d$ and notice that $\mathbf{r}_m = \mathbf{p}$.

$$\Delta\mathbf{m}_{\text{good}} \quad = \quad \mathbf{A}^{-1}(\mathbf{A}^{-1})' \, \mathbf{g} + \mathbf{m} \tag{5.17}$$

The search direction (5.17) shows a positive-definite operator scaling the gradient. Each component of any gradient vector is independent of each other. All independently point a direction for descent. Obviously, each can be scaled by any positive number. Now we have found that we can also scale a gradient vector by a positive definite matrix and we can still expect the conjugate-direction algorithm to descend, as always, to the "exact" answer in a finite number of steps. This is because modifying the search direction with $\mathbf{A}^{-1}(\mathbf{A}^{-1})'$ is equivalent to solving a conjugate-gradient problem in $\mathbf{p}$.

5.4. NULL SPACE AND INTERVAL VELOCITY

A bread-and-butter problem in seismology is building the velocity as a function of depth (or vertical travel time) starting from certain measurements. The measurements are described elsewhere (BEI for example). They amount to measuring the integral of the velocity squared from the surface down to the reflector. It is known as

the RMS (root-mean-square) velocity. Although good quality echos may arrive often, they rarely arrive continuously for all depths. Good information is interspersed unpredictably with poor information. Luckily we can also estimate the data quality by the "coherency" or the "stack energy". In summary, what we get from observations and preprocessing are two functions of travel-time depth, (1) the integrated (from the surface) squared velocity, and (2) a measure of the quality of the integrated velocity measurement. Some definitions:

d is a data vector whose components range over the vertical traveltime depth τ, and whose component values contain the scaled RMS velocity squared $\tau v_{\text{RMS}}^2/\Delta\tau$ where $\tau/\Delta\tau$ is the index on the time axis.

W is a diagonal matrix along which we lay the given measure of data quality. We will use it as a weighting function.

C is the matrix of causal integration, a lower triangular matrix of ones.

D is the matrix of causal differentiation, namely, $\mathbf{D} = \mathbf{C}^{-1}$.

u is a vector whose components range over the vertical traveltime depth τ, and whose component values contain the interval velocity squared v_{interval}^2.

From these definitions, under the assumption of a stratified earth with horizontal reflectors (and no multiple reflections) the theoretical (squared) interval velocities enable us to define the theoretical (squared) RMS velocities by

$$\mathbf{Cu} \quad = \quad \mathbf{d} \tag{5.18}$$

With imperfect data, our data fitting goal is to minimize the residual

$$\mathbf{0} \quad \approx \quad \mathbf{W}[\mathbf{Cu} - \mathbf{d}] \tag{5.19}$$

To find the interval velocity where there is no data (where the stack power theoretically vanishes) we have the "model damping" goal to minimize the wiggliness **p** of the squared interval velocity **u**.

$$\mathbf{0} \quad \approx \quad \mathbf{Du} \quad = \quad \mathbf{p} \tag{5.20}$$

We precondition these two goals by changing the optimization variable from interval velocity squared **u** to its wiggliness **p**. Substituting $\mathbf{u} = \mathbf{Cp}$ gives the two goals expressed as a function of wiggliness **p**.

$$\mathbf{0} \quad \approx \quad \mathbf{W}\left[\mathbf{C}^2\mathbf{p} - \mathbf{d}\right] \tag{5.21}$$

$$\mathbf{0} \quad \approx \quad \epsilon\, \mathbf{p} \tag{5.22}$$

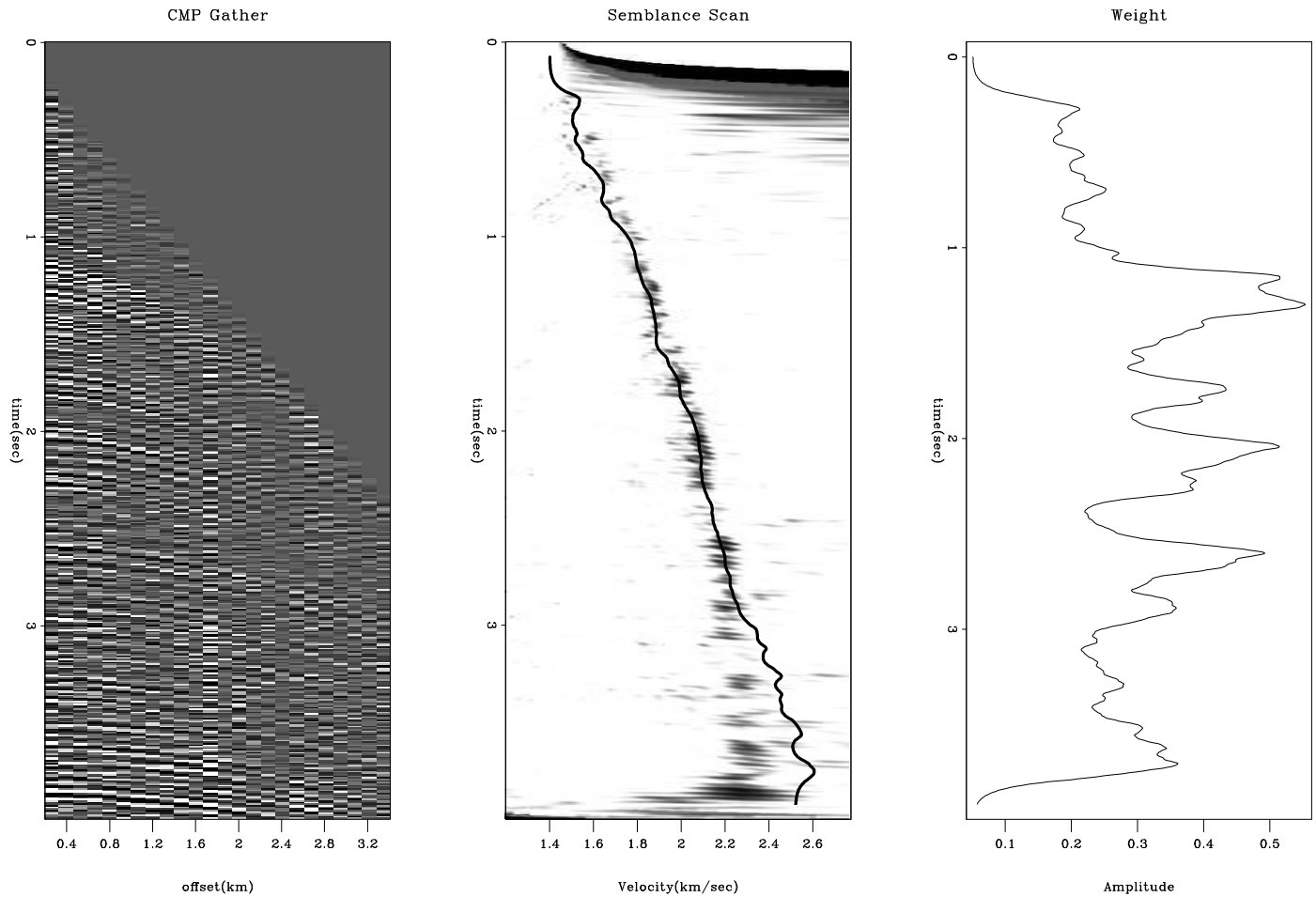


Figure 5.1: Raw CMP gather (left), Semblance scan (middle), and semblance value used for weighting function (right). (Clapp) prc-clapp [ER,M]

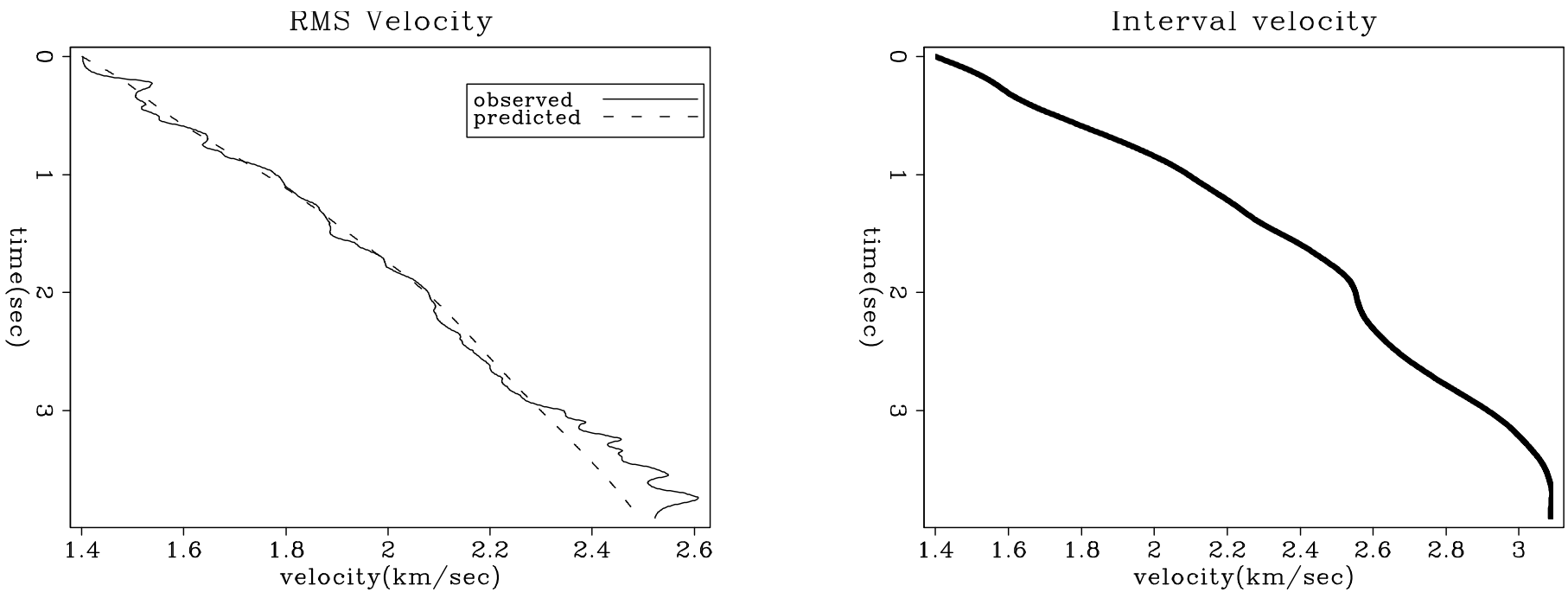


Figure 5.2: Observed RMS velocity and that predicted by a stiff model with $\epsilon = 4$. (Clapp) prc-stiff [ER]

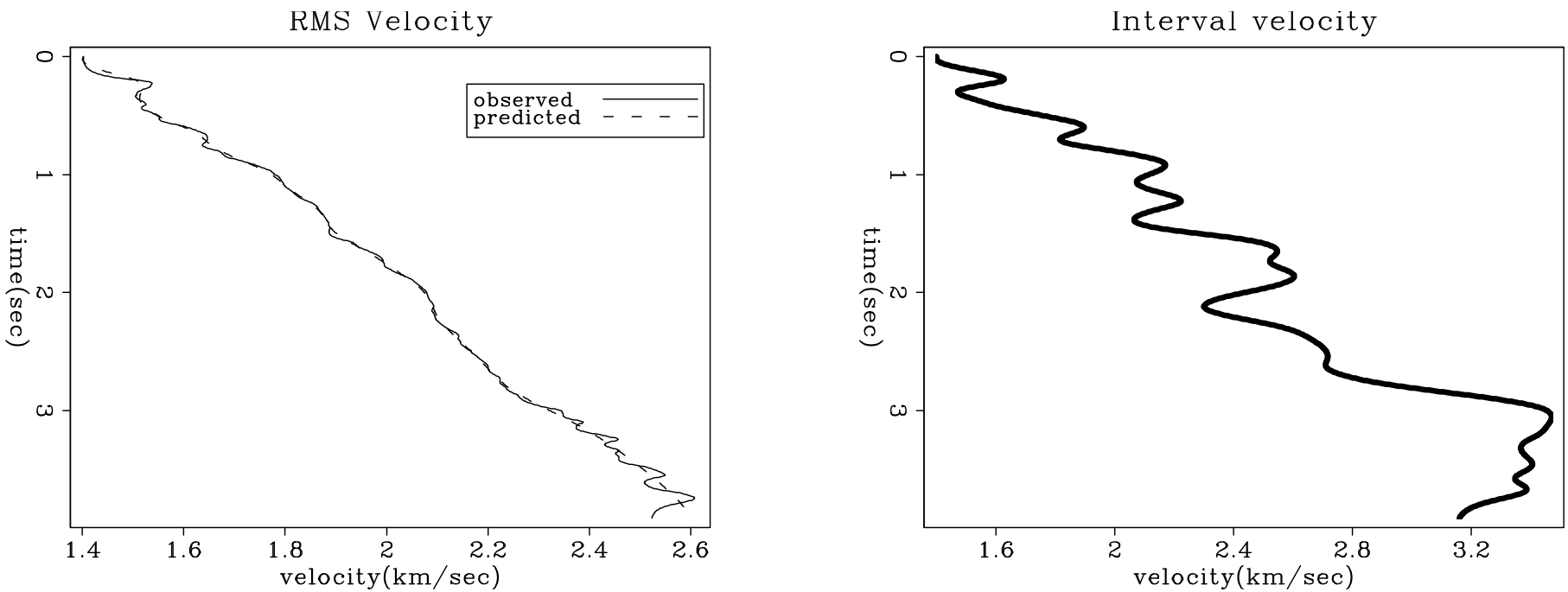


Figure 5.3: Observed RMS velocity and that predicted by a flexible model with $\epsilon = .25$ (Clapp) prc-flex [ER]

```
module vrms2int_mod {                # Transform from RMS to interval velocity
  use causint
  use weight
  use mask1
  use cgstep_mod
  use solver_prc_mod
contains
  subroutine vrms2int( niter, eps, weight, vrms, vint) {
    integer,             intent( in)     :: niter     # iterations
    real,                intent( in)     :: eps       # scaling parameter
    real, dimension(:), intent( in out) :: vrms      # RMS velocity
    real, dimension(:), intent( out)    :: vint      # interval velocity
    real, dimension(:), pointer         :: weight    # data weighting
    integer                             :: st,it,nt
    logical, dimension( size( vint))    :: mask
    logical, dimension(:), pointer      :: msk
    real,    dimension( size( vrms))    :: dat ,wt
    real, dimension(:), pointer         :: wght
    nt = size( vrms)
    do it= 1, nt {
       dat( it) = vrms( it) * vrms( it) * it
       wt( it) = weight( it)*(1./it)           # decrease weight with time
       }
    mask = .false.;   mask( 1) = .true.            # constrain first point
    vint = 0.     ;   vint( 1) = dat( 1)
    allocate(wght(size(wt)))
    wght=wt
    call weight_init(wght)
    allocate(msk(size(mask)))
    msk=.not.mask
    call mask1_init(msk)
    call solver_prc( m=vint, d=dat,Fop=causint_lop, stepper=cgstep, niter=niter, &
        Sop= causint_lop, nSop=nt, eps = eps,verb=.true.,Jop=mask1_lop, &
        p0=vint, Wop=weight_lop)
    call cgstep_close()
    st = causint_lop( .false., .false., vint, dat)
    do it= 1, nt
       vrms( it) = sqrt( dat( it)/it)
```

5.4.1. Balancing good data with bad

Choosing the size of ϵ chooses the stiffness of the curve that connects regions of good data. Our first test cases gave solutions that we interpreted to be too stiff at early times and too flexible at later times. This leads to two possible ways to deal with the problem. One way modifies the model shaping and the other modifies the data fitting. The program below weakens the data fitting weight with time. This has the same effect as stiffening the model shaping with time. vrms2int

5.4.2. Lateral variations

The analysis above appears one dimensional in depth. Conventional interval velocity estimation builds a velocity-depth model independently at each lateral location. Here we have a logical path for combining measurements from various lateral locations. We can change the regularization to something like $\mathbf{0} \approx \nabla \mathbf{u}$. Instead of merely minimizing the vertical gradient of velocity we minimize its spatial gradient. Luckily we have preconditioning and the helix to speed the solution.

5.4.3. Blocky models

Sometimes we seek a velocity model that increases smoothly with depth through our scattered measurements of good-quality RMS velocities. Other times, we seek a blocky model. (Where seismic data is poor, a well log could tell us whether to choose smooth or blocky.) Here we see an estimation method that can choose the blocky alternative, or some combination of smooth and blocky.

Consider the five layer model in Figure 5.4. Each layer has unit traveltime thickness (so integration is simply summation). Let the squared interval velocities be (a,b,c,d,e) with strong reliable reflections at the base of layer c and layer e, and weak, incoherent, "bad" reflections at bases of (a,b,d). Thus we measure V_c^2 the RMS velocity squared of the top three layers and V_e^2 that for all five layers. Since we have no reflection from at the base of the fourth layer, the velocity in the fourth layer is not measured but a matter for choice. In a smooth linear fit we would want $d = (c+e)/2$. In a blocky fit we would want $d = e$.

Our screen for good reflections looks like $(0,0,1,0,1)$ and our screen for bad ones looks like the complement $(1,1,0,1,0)$. We put these screens on the diagonals

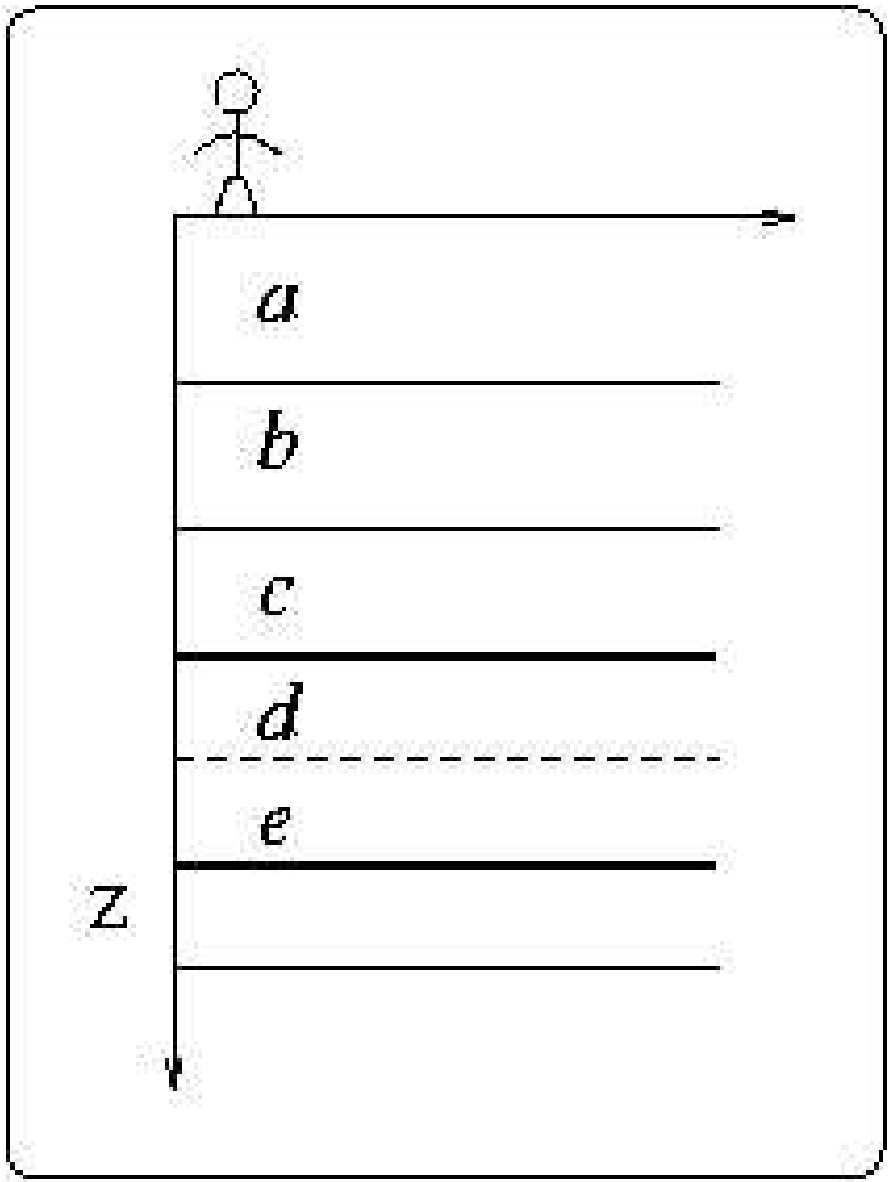


Figure 5.4: A layered earth model. The layer interfaces cause reflections. Each layer has a constant velocity in its interior. prc-rosales [NR]

of diagonal matrices **G** and **B**. Our fitting goals are:

$$3V_c^2 \approx a+b+c \tag{5.23}$$
$$5V_e^2 \approx a+b+c+d+e \tag{5.24}$$
$$u_0 \approx a \tag{5.25}$$
$$0 \approx -a+b \tag{5.26}$$
$$0 \approx -b+c \tag{5.27}$$
$$0 \approx -c+d \tag{5.28}$$
$$0 \approx -d+e \tag{5.29}$$

For the blocky solution, we do not want the fitting goal (5.28). Further explanations await completion of examples.

5.5. INVERSE LINEAR INTERPOLATION

The first example is a simple synthetic test for 1-D inverse interpolation. The input data were randomly subsampled (with decreasing density) from a sinusoid (Figure

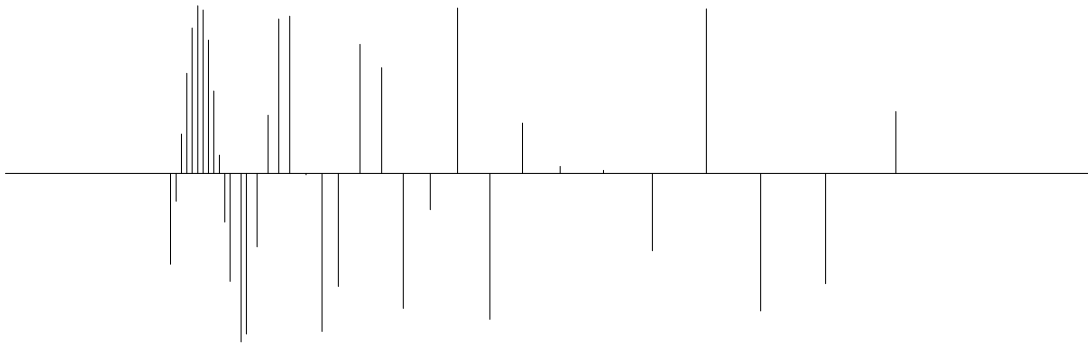


Figure 5.5: The input data are irregularly sampled. prc-data [ER]

5.5). The forward operator **L** in this case is linear interpolation. We seek a regularly sampled model that could predict the data with a forward linear interpolation. Sparse irregular distribution of the input data makes the regularization enforcement a necessity. I applied convolution with the simple $(1, -1)$ difference filter as the operator **D** that forces model continuity (the first-order spline). An appropriate preconditioner **S** in this case is recursive causal integration. As expected, preconditioning provides a much faster rate of convergence. Since iteration to the exact solution is never achieved in large-scale problems, the results of iterative optimization may turn out quite differently. Bill Harlan points out that the two goals in (5.6.1) conflict with each other: the first one enforces "details" in the model, while the second one tries to smooth them out. Typically, regularized optimization creates a compli-

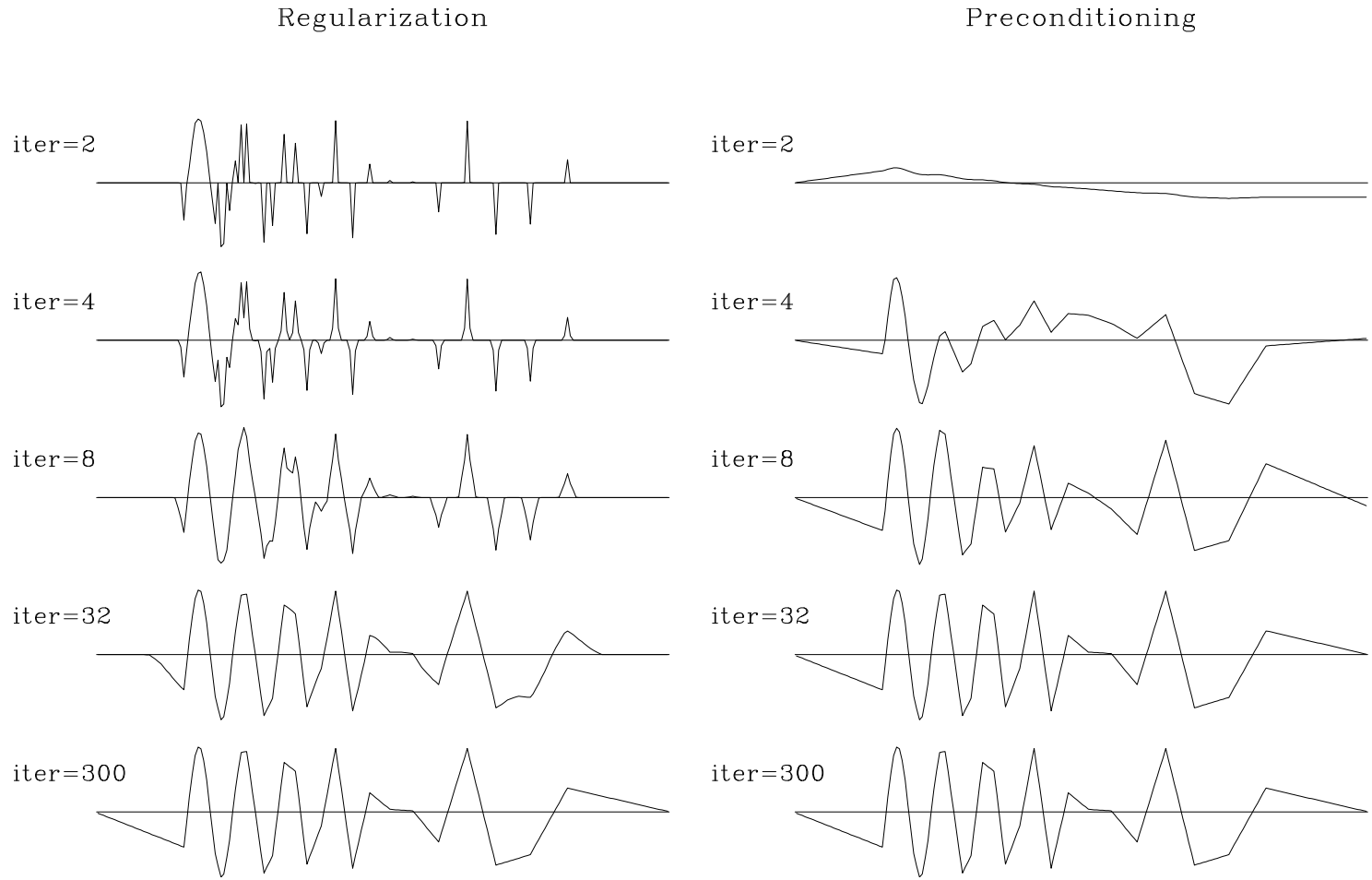


Figure 5.6: Convergence history of inverse linear interpolation. Left: regularization, right: preconditioning. The regularization operator **A** is the derivative operator (convolution with $(1, -1)$. The preconditioning operator **S** is causal integration. prc-conv1 [ER,M]

cated model at early iterations. At first, the data fitting goal (5.6.1) plays a more important role. Later, the regularization goal (5.6.1) comes into play and simplifies (smooths) the model as much as needed. Preconditioning acts differently. The very first iterations create a simplified (smooth) model. Later, the data fitting goal adds more details into the model. If we stop the iterative process early, we end up with an insufficiently complex model, not in an insufficiently simplified one. Figure 5.6 provides a clear illustration of Harlan's observation.

Figure 5.7 measures the rate of convergence by the model residual, which is a distance from the current model to the final solution. It shows that preconditioning saves many iterations. Since the cost of each iteration for each method is roughly equal, the efficiency of preconditioning is evident.

The module `invint2` /prog:invint2 invokes the solvers to make Figures 5.6 and 5.7. We use convolution with `helicon` /prog:helicon for the regularization and we use deconvolution with `polydiv` /prog:polydiv for the preconditioning. The code looks fairly straightforward except for the oxymoron `known=aa%mis`. invint2

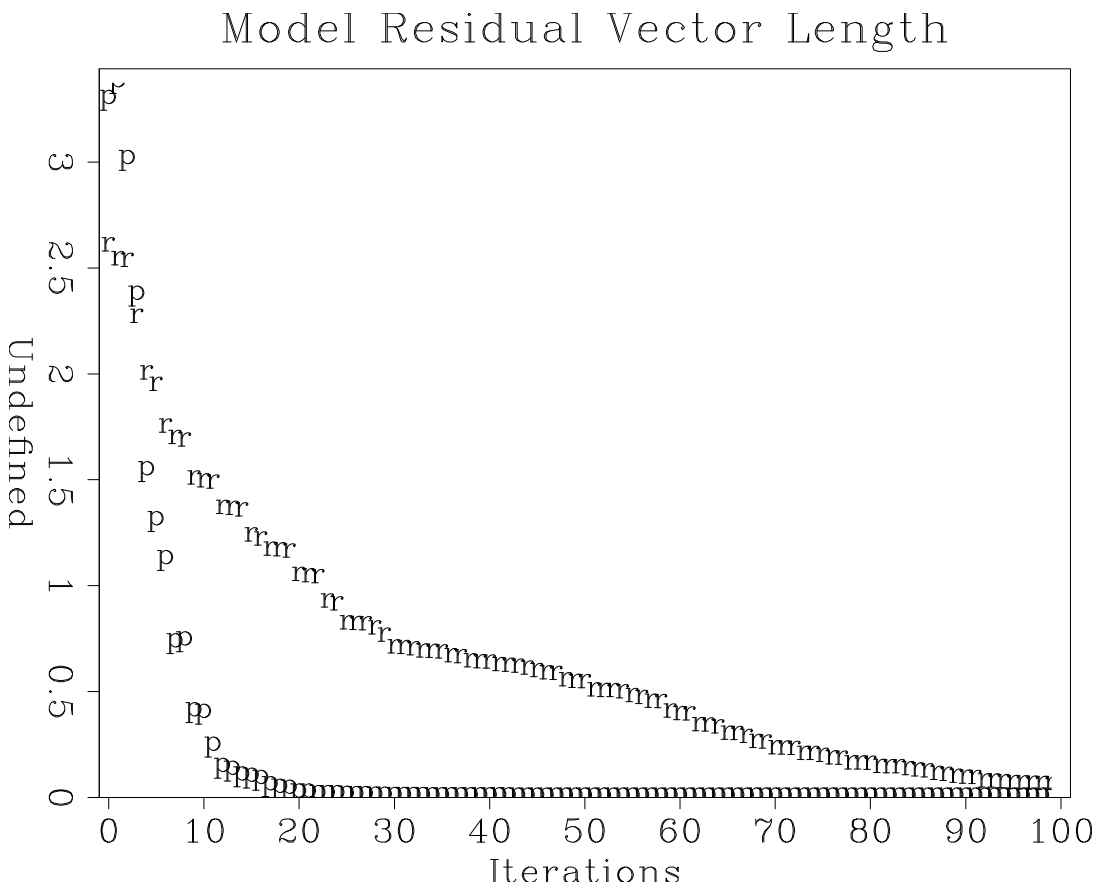


Figure 5.7: Convergence of the iterative optimization, measured in terms of the model residual. The "p" points stand for preconditioning; the "r" points, regularization. prc-schwab1 [ER]

```
module invint2 {   # Inverse linear interpolation
  use lint1
  use helicon                          # regularized    by helix   filtering
  use polydiv                          # preconditioned by inverse filtering
  use cgstep_mod
  use solver_reg_mod
  use solver_prc_mod
contains
  subroutine invint( niter, coord,ord, o1,d1, mm,mmov, eps, aa, method) {
    logical,                  intent( in)  :: method
    integer,                  intent( in)  :: niter
    real,                     intent( in)  :: o1, d1, eps
    real,    dimension( :),   intent( in)  :: ord
    type( filter),            intent( in)  :: aa
    real,    dimension( :),   intent( out) :: mm
    real,    dimension( :,:), intent( out) :: mmov         # model movie
    real,    dimension( :),   pointer      :: coord        # coordinate
    call lint1_init( o1, d1, coord)
    if( method) {                                          # preconditioning
       call polydiv_init( size(mm), aa)
       call solver_prc( Fop=lint1_lop, stepper=cgstep, niter=niter, m=mm, d=ord,
                        Sop=polydiv_lop, nSop=size(mm), eps=eps, mmov=mmov, verb=.true.
       call polydiv_close()
    } else {                                               # regularization
       call helicon_init( aa)
       call solver_reg( Fop=lint1_lop, stepper=cgstep, niter=niter, m=mm, d=ord,
                        Aop=helicon_lop, nAop=size(mm), eps=eps, mmov=mmov, verb=.true.
    }
    call cgstep_close()
  }
}
```

Back

5.6. EMPTY BINS AND PRECONDITIONING

There are at least three ways to fill empty bins. Two require a roughening operator $\mathbf{A}$ while the third requires a smoothing operator which (for comparison purposes) we denote $\mathbf{A}^{-1}$. The three methods are generally equivalent though they differ in important details.

The original way in Chapter 3 is to restore missing data by ensuring that the restored data, after specified filtering, has minimum energy, say $\mathbf{Am} \approx \mathbf{0}$. Introduce the selection mask operator $\mathbf{K}$, a diagonal matrix with ones on the known data and zeros elsewhere (on the missing data). Thus $\mathbf{0} \approx \mathbf{A}(\mathbf{I}-\mathbf{K}+\mathbf{K})\mathbf{m}$ or

$$\mathbf{0} \quad \approx \quad \mathbf{A}(\mathbf{I}-\mathbf{K})\mathbf{m} \;+\; \mathbf{A}\mathbf{m}_k \;, \tag{5.30}$$

where we define $\mathbf{m}_k$ to be the data with missing values set to zero by $\mathbf{m}_k = \mathbf{Km}$.

A second way to find missing data is with the set of goals

$$\begin{aligned} \mathbf{0} &\approx \mathbf{Km} - \mathbf{m}_k \\ \mathbf{0} &\approx \epsilon\mathbf{Am} \end{aligned} \tag{5.31}$$

and take the limit as the scalar $\epsilon \to 0$. At that limit, we should have the same result as equation (5.30).

There is an important philosophical difference between the first method and the second. The first method strictly honors the known data. The second method acknowledges that when data misfits the regularization theory, it might be the fault of the data so the data need not be strictly honored. Just what balance is proper falls to the numerical choice of ϵ, a nontrivial topic.

A third way to find missing data is to precondition equation (5.31), namely, try the substitution $\mathbf{m} = \mathbf{A}^{-1}\mathbf{p}$.

$$\begin{array}{lcrcl} \mathbf{0} & \approx & \mathbf{K}\mathbf{A}^{-1}\mathbf{p} & - & \mathbf{m}_k \\ \mathbf{0} & \approx & \epsilon\mathbf{p} & & \end{array} \tag{5.32}$$

There is no simple way of knowing beforehand what is the best value of ϵ. Practitioners like to see solutions for various values of ϵ. Of course that can cost a lot of computational effort. Practical exploratory data analysis is more pragmatic. Without a simple clear theoretical basis, analysts generally begin from $\mathbf{p} = 0$ and abandon the fitting goal $\epsilon\mathbf{I}\mathbf{p} \approx 0$. Implicitly, they take $\epsilon = 0$. Then they examine the solution as a function of iteration, imagining that the solution at larger iterations corresponds to smaller ϵ. There is an eigenvector analysis indicating some kind of basis for this approach, but I believe there is no firm guidance.

Before we look at coding details for the three methods of filling the empty bins, we'll compare results of trying all three methods. For the roughening operator $\mathbf{A}$, we'll take the helix derivative $\mathbf{H}$. This is logically equivalent to roughening with the gradient ∇ because the (negative) laplacian operator is $\nabla' \nabla = \mathbf{H}' \mathbf{H}$.

5.6.1. Faking the epsilon

I'll tell a common story that should horrify those interested in theory (under 23 years old), and delight those wanting to get real work done (over 23 years old). Suppose you have a problem where model space is an image, maybe 1000×1000, while application of the operator $\mathbf{F}$ takes about an hour. A common example in petroleum prospecting is "wave equation migration". The older generation of workers thus did a migration (single application of an adjoint operator) and finished their task in an hour. You, however, have read this book or otherwise learned about inversion (iteratively modeling for an increasingly better fitting model). Twelve iterations (12 applications of both $\mathbf{F}'$ and $\mathbf{F}$) requires 24 hours. Theoretically you should be doing a million interations (10^6, one for each point in model space), but you are hoping 12 will be enough. (Good luck!)

Oops! You also need to choose a numerical value for ϵ. Since we have no theory for that choice, perhaps we should scan over 7 possible values, or maybe 30. Now the job takes a week or a month. That's really bad news because there typically are other more important and more difficult problems we also need to be working out (such as updating the seismic velocity model). Any trick, honest or sly, would be nice to know. I'll tell you a trick that is very widely used.

Imagine we had plenty of computer time. For small ϵ we have a smaller residual of data fitting but a larger residual in the model styling goal. Conversely, for large ϵ, we may fit the data poorly, but the model styling goal is well honored. Now please carefully examine the preconditioned steps in Figure 5.6. At early iterations we see the solution is piece-wise lines (good) but many data points are poorly fit (bad). This is like large ϵ. After more iterations the solution fits the data increasingly well. This is like small ϵ. Are you ready to guess what pragmatic people do when they are pressured to get results?

Many (most?) practical studies are done ignoring (abandoning) the model styling regression (second fitting regression below):

$$\begin{aligned} \mathbf{0} &\approx \mathbf{F}\mathbf{A}^{-1}\mathbf{p} - \mathbf{d} \\ \mathbf{0} &\approx \epsilon\mathbf{p} \end{aligned}$$

Nice to be rid of ϵ! The pragmatic person iterates the data fitting regression only, watches the solution as a function of iteration, and stops when tired, or (more hopefully) stops at the iteration that is subjectively most pleasing.

5.6.2. SEABEAM: Filling the empty bins with a laplacian

Figure 5.8 shows a day's worth of data[1] collected at sea by SeaBeam, an apparatus for measuring water depth both directly under a ship, and somewhat off to the sides of the ship's track. The data is measurements of depth $h(x, y)$ at miscellaneous locations in the (x, y)-plane. The locations are scattered about, according to various aspects of the ship's navigation and the geometry of the SeaBeam sonic antenna. Figure 5.8 was made by binning with `bin2()` /prog:bin2 and equation (1.15). The spatial spectra of the noise in the data could be estimated where tracks cross over themselves. This might be worth while, but we do not pursue it now.

Here we focus on the empty mesh locations where no data is recorded (displayed with the value of the mean depth $\overline{h}$). These empty bins were filled with module `mis2` /prog:mis2. Results are in Figure 5.9. In Figure 5.9 the left column

[1] I'd like to thank Alistair Harding for this interesting data set named April 18.

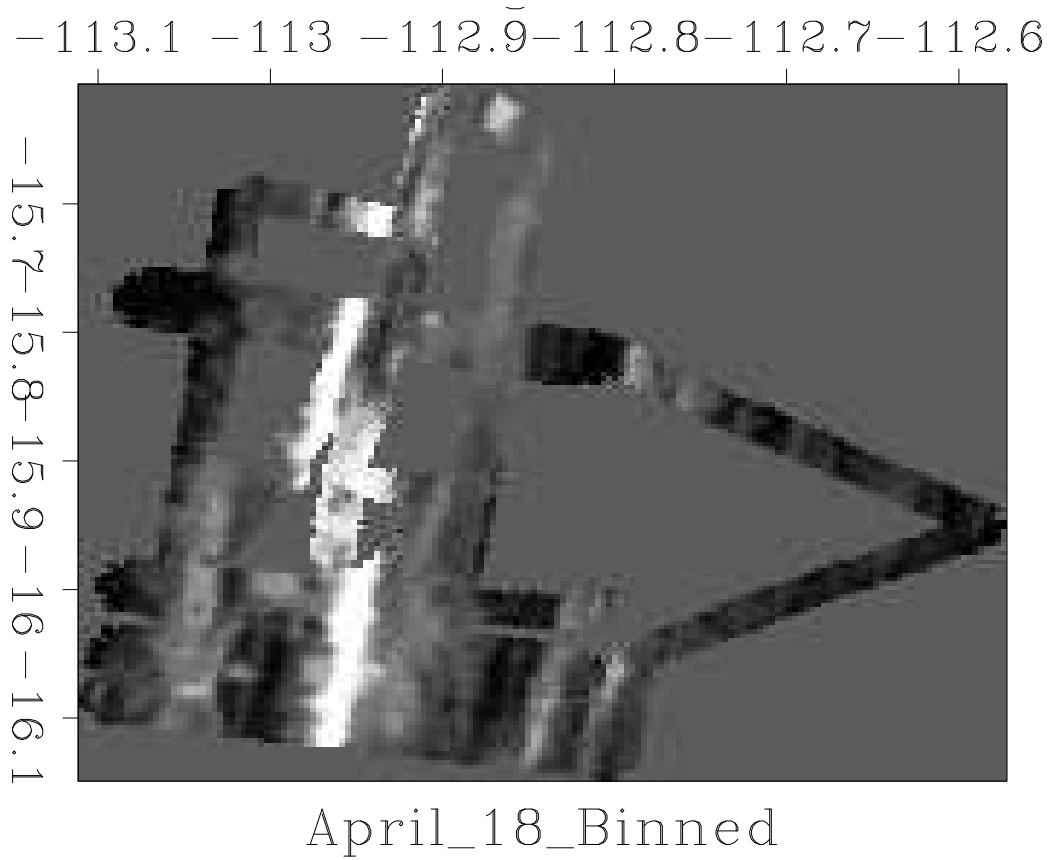


Figure 5.8: Depth of the ocean under ship tracks. Empty bins are displayed with an average depth $\bar{h}$. prc-seabin90 [ER]

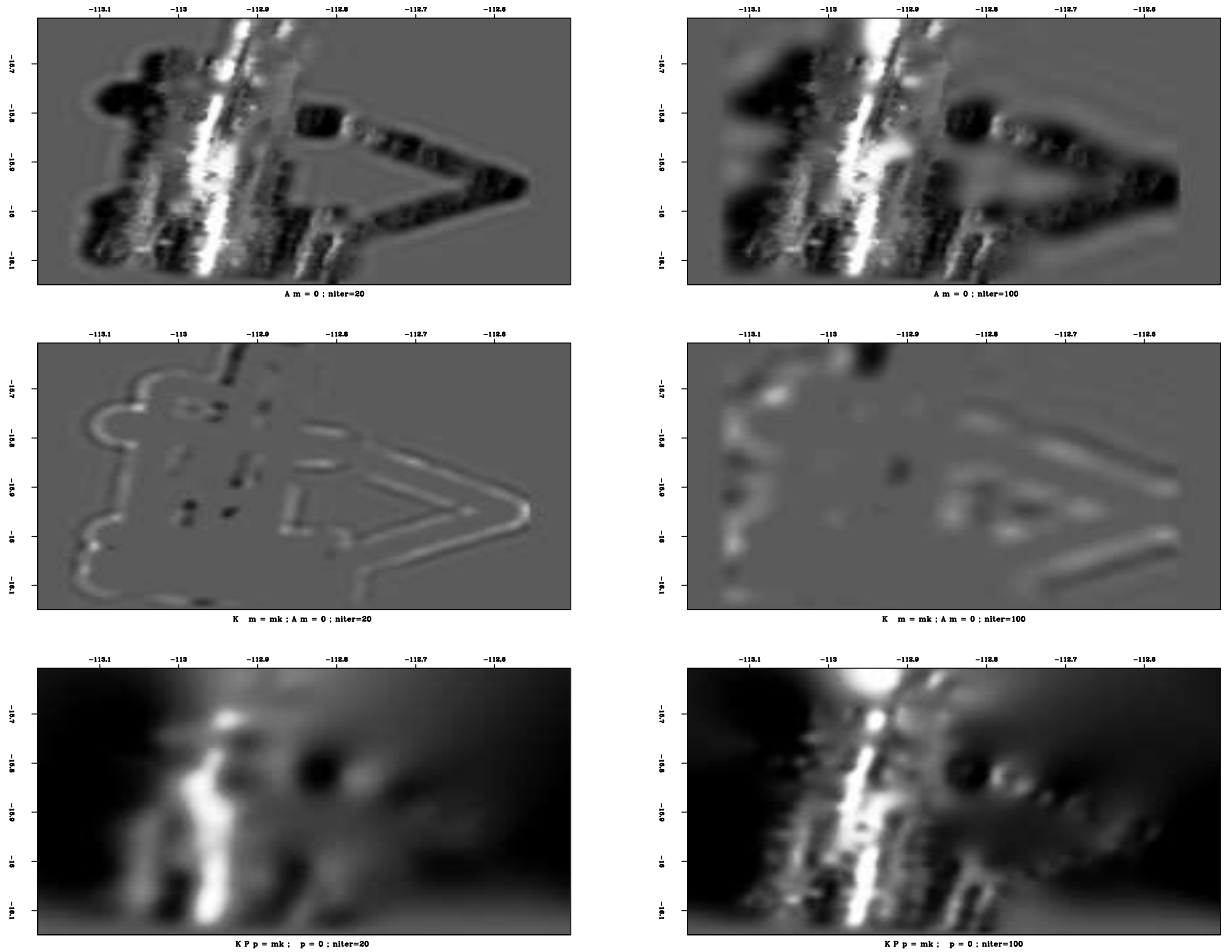


Figure 5.9: The ocean bottom restoring missing data with a helix derivative. prc-prcfill [ER,M]

results from 20 iterations while the right column results from 100 iterations.

The top row in Figure 5.9 shows that more iterations spreads information further into the region of missing data.

It turned out that the original method strictly honoring known data gave results so similar to the second method (regularizing) that the plots could not be visually distinguished. The middle row in Figure 5.9 therefore shows the difference in the result of the two methods. We see an outline of the transition between known and unknown regions. Obviously, the missing data is pulling known data towards zero.

The bottom row in Figure 5.9 shows that preconditioning spreads information to great distances much quicker but early iterations make little effort to honor the data. (Even though these results are for $\epsilon = 0$.) Later iterations make little change at long distance but begin to restore sharp details on the small features of the known topography.

What if we can only afford 100 iterations? Perhaps we should first do 50 iterations with preconditioning to develop the remote part of the solution and then do 50 iterations by one of the other two methods to be sure we attended to the details near the known data. A more unified approach (not yet tried, as far as I know) would be to unify the techniques. The conjugate direction method searches two directions, the

gradient and the previous step. We could add a third direction, the smart direction of equation (5.17). Instead of having a 2×2 matrix solution like equation (2.80) for two distances, we would need to solve a 3×3 matrix for three.

Figure 5.9 has a few artifacts connected with the use of the helix derivative. Examine equation (4.11) to notice the shape of the helix derivative. In principle, it is infinitely long in the horizontal axis in both equation (4.11) and Figure 5.9. In practice, it is truncated. The truncation is visible as bands along the sides of Figure 5.9.

As a practical matter, no one would use the first two bin filling methods with helix derivative for the roughener because it is theoretically equivalent to the gradient operator ∇ which has many fewer coefficients. Later, in Chapter 6 we'll find a much smarter roughening operator $\mathbf{A}$ called the Prediction Error Filter (PEF) which gives better results.

5.6.3. Three codes for inverse masking

The selection (or masking) operator $\mathbf{K}$ is implemented in `mask1()` /prog:mask1.

mask1 All the results shown in Figure 5.9 were created with the module `mis2`

```
module mask1 { # masking operator
  logical, dimension( :), pointer :: m
#% _init( m)
#% _lop( x, y)
  if( adj)
            where( m) x += y
  else  #
            where( m) y += x
}
```

Back

```
module mis2_mod {
  use mask1 + helicon + polydiv + cgstep_mod
  use solver_smp_mod + solver_reg_mod + solver_prc_mod
contains
  subroutine mis2( niter, xx, aa, known, style) {
    integer,                intent( in)     :: style
    integer,                intent( in)     :: niter
    type( filter),          intent( in)     :: aa
    logical, dimension( :), intent( in)     :: known
    real,    dimension( :), intent( in out) :: xx       # fitting variables
    real,    dimension( :), allocatable     :: dd
    logical, dimension( :), pointer         :: msk
    integer                                 :: nx
    nx = size( xx)
    allocate( dd(nx))
    allocate(msk(nx))
    if(style==0) {
            dd = 0.
            msk = .not.known
            call mask1_init( msk)
            call helicon_init( aa)
            call solver_smp( m=xx, d=dd, Fop=helicon_lop, stepper=cgstep,
                niter=niter, Jop=mask1_lop, m0=xx)
            call helicon_close()
    } else if(style==1) {
            dd=xx
            msk = known
            call mask1_init( msk)
            call helicon_init( aa)
            call solver_reg( m=xx, d=dd, Fop=mask1_lop, stepper=cgstep,
                niter=niter, Aop=helicon_lop, nAop=nx, eps=0.1)
            call helicon_close()
    } else {
            dd=xx
            msk = known
            call mask1_init( msk)
            call polydiv_init( nx, aa)
            call solver_prc( m=xx, d=dd, Fop=mask1_lop, stepper=cgstep,
```

/prog:mis2. Code locations with `style=0,1,2` correspond to the fitting goals (5.30), (5.31), (5.32). mis2

5.7. THEORY OF UNDERDETERMINED LEAST-SQUARES

Construct theoretical data with

$$\mathbf{d} \quad = \quad \mathbf{F}\mathbf{m} \tag{5.33}$$

Assume there are fewer data points than model points and that the matrix $\mathbf{F}\mathbf{F}'$ is invertible. From the theoretical data we estimate a model $\mathbf{m}_0$ with

$$\mathbf{m}_0 \quad = \quad \mathbf{F}'(\mathbf{F}\mathbf{F}')^{-1}\mathbf{d} \tag{5.34}$$

To verify the validity of the estimate, insert the estimate (5.34) into the data modeling equation (5.33) and notice that the estimate $\mathbf{m}_0$ predicts the correct data. Notice that equation (5.34) is not the same as equation (2.40) which we derived much earlier. What's the difference? The central issue is which matrix of $\mathbf{F}\mathbf{F}'$ and $\mathbf{F}'\mathbf{F}$

actually has an inverse. If **F** is a rectangular matrix, then it is certain that one of the two is not invertible. (There are plenty of real cases where neither matrix is invertible. That's one reason we use iterative solvers.) Here we are dealing with the case with more model points than data points.

Now we will show that of all possible models **m** that predict the correct data, $\mathbf{m}_0$ has the least energy. (I'd like to thank Sergey Fomel for this clear and simple proof that does *not* use Lagrange multipliers.) First split (5.34) into an intermediate result $\mathbf{d}_0$ and final result:

$$\mathbf{d}_0 = (\mathbf{F}\mathbf{F}')^{-1}\mathbf{d} \tag{5.35}$$

$$\mathbf{m}_0 = \mathbf{F}'\mathbf{d}_0 \tag{5.36}$$

Consider another model (**x** not equal to zero)

$$\mathbf{m} = \mathbf{m}_0 + \mathbf{x} \tag{5.37}$$

which fits the theoretical data $\mathbf{d} = \mathbf{F}(\mathbf{m}_0 + \mathbf{x})$. Since $\mathbf{d} = \mathbf{F}\mathbf{m}_0$, we see that **x** is a null space vector.

$$\mathbf{F}\mathbf{x} = \mathbf{0} \tag{5.38}$$

First we see that $\mathbf{m}_0$ is orthogonal to $\mathbf{x}$ because

$$\mathbf{m}_0'\mathbf{x} \quad = \quad (\mathbf{F}'\mathbf{d}_0)'\mathbf{x} \quad = \quad \mathbf{d}_0'\mathbf{F}\mathbf{x} \quad = \quad \mathbf{d}_0'\mathbf{0} \quad = \quad 0 \tag{5.39}$$

Therefore,

$$\mathbf{m}'\mathbf{m} \quad = \quad \mathbf{m}_0'\mathbf{m}_0 + \mathbf{x}'\mathbf{x} + 2\mathbf{x}'\mathbf{m}_0 \quad = \quad \mathbf{m}_0'\mathbf{m}_0 + \mathbf{x}'\mathbf{x} \quad \geq \quad \mathbf{m}_0'\mathbf{m}_0 \tag{5.40}$$

so adding null space to $\mathbf{m}_0$ can only increase its energy. In summary, the solution $\mathbf{m}_0 = \mathbf{F}'(\mathbf{F}\mathbf{F}')^{-1}\mathbf{d}$ has less energy than any other model that satisfies the data.

Not only does the theoretical solution $\mathbf{m}_0 = \mathbf{F}'(\mathbf{F}\mathbf{F}')^{-1}\mathbf{d}$ have minimum energy, but the result of iterative descent will too, provided that we begin iterations from $\mathbf{m}_0 = 0$ or any $\mathbf{m}_0$ with no null-space component. In (5.39) we see that the orthogonality $\mathbf{m}_0'\mathbf{x} = 0$ does not arise because $\mathbf{d}_0$ has any particular value. It arises because $\mathbf{m}_0$ is of the form $\mathbf{F}'\mathbf{d}_0$. Gradient methods contribute $\Delta\mathbf{m} = \mathbf{F}'\mathbf{r}$ which is of the required form.

5.8. SCALING THE ADJOINT

First I remind you of a rarely used little bit of mathematical notation. Given a vector $\mathbf{m}$ with components (m_1, m_2, m_3), the notation **diag m** means

$$\mathbf{diag\ m} \quad = \quad \begin{bmatrix} m_1 & 0 & 0 \\ 0 & m_2 & 0 \\ 0 & 0 & m_3 \end{bmatrix} \tag{5.41}$$

Given the usual linearized fitting goal between data space and model space, $\mathbf{d} \approx \mathbf{Fm}$, the simplest image of the model space results from application of the adjoint operator $\hat{\mathbf{m}} = \mathbf{F}'\mathbf{d}$. Unless $\mathbf{F}$ has no physical units, however, the physical units of $\hat{\mathbf{m}}$ do not match those of $\mathbf{m}$, so we need a scaling factor. The theoretical solution $\mathbf{m}_{\text{theor}} = (\mathbf{F}'\mathbf{F})^{-1}\mathbf{F}'\mathbf{d}$ tells us that the scaling units should be those of $(\mathbf{F}'\mathbf{F})^{-1}$. We are going to approximate $(\mathbf{F}'\mathbf{F})^{-1}$ by a diagonal matrix $\mathbf{W}^2$ with the correct units so $\hat{\mathbf{m}} = \mathbf{W}^2\mathbf{F}'\mathbf{d}$.

What we use for $\mathbf{W}$ will be a guess, simply a guess. If it works better than nothing, we'll be happy, and if it doesn't we'll forget about it. Experience shows it is a good idea to try. Common sense tells us to insist that all elements of $\mathbf{W}^2$ are positive. $\mathbf{W}^2$ is a square matrix of size of model space. From any vector $\tilde{\mathbf{m}}$ in

model space with all positive components, we could guess that $\mathbf{W}^2$ be $\mathbf{diag}\ \tilde{\mathbf{m}}$ to any power. To get the right physical dimensions we choose $\tilde{\mathbf{m}} = \mathbf{1}$, a vector of all ones and choose

$$\mathbf{W}^2 \quad = \quad \frac{1}{\mathbf{diag}\ \mathbf{F}'\mathbf{F}\mathbf{1}} \tag{5.42}$$

A problem with the choice (5.42) is that some components might be zero or negative. Well, we can take the square root of the squares of components and/or smooth the result.

To go beyond the scaled adjoint we can use $\mathbf{W}$ as a preconditioner. To use $\mathbf{W}$ as a preconditioner we define implicitly a new set of variables $\mathbf{p}$ by the substitution $\mathbf{m} = \mathbf{W}\mathbf{p}$. Then $\mathbf{d} \approx \mathbf{F}\mathbf{m} = \mathbf{F}\mathbf{W}\mathbf{p}$. To find $\mathbf{p}$ instead of $\mathbf{m}$, we iterate with the operator $\mathbf{F}\mathbf{W}$ instead of with $\mathbf{F}$. As usual, the first step of the iteration is to use the adjoint of $\mathbf{d} \approx \mathbf{F}\mathbf{W}\mathbf{p}$ to form the image $\hat{\mathbf{p}} = (\mathbf{F}\mathbf{W})'\mathbf{d}$. At the end of the iterations, we convert from $\mathbf{p}$ back to $\mathbf{m}$ with $\mathbf{m} = \mathbf{W}\mathbf{p}$. The result after the first iteration $\hat{\mathbf{m}} = \mathbf{W}\hat{\mathbf{p}} = \mathbf{W}(\mathbf{F}\mathbf{W})'\mathbf{d} = \mathbf{W}^2\mathbf{F}'\mathbf{d}$ turns out to be the same as scaling.

By (5.42), $\mathbf{W}$ has physical units inverse to $\mathbf{F}$. Thus the transformation $\mathbf{F}\mathbf{W}$ has no units so the $\mathbf{p}$ variables have physical units of data space. Experimentalists might enjoy seeing the solution $\mathbf{p}$ with its data units more than viewing the solution $\mathbf{m}$ with

its more theoretical model units.

The theoretical solution for underdetermined systems $\mathbf{m} = \mathbf{F}'(\mathbf{F}\mathbf{F}')^{-1}\mathbf{d}$ suggests an alternate approach using instead $\hat{\mathbf{m}} = \mathbf{F}'\mathbf{W}_d^2\mathbf{d}$. This diagonal weighting matrix $\mathbf{W}_d^2$ must be drawn from vectors in data space. Again I chose a vector of all 1's getting the weight

$$\mathbf{W}_d^2 \quad = \quad \frac{1}{\mathbf{diag}\ \mathbf{F}\mathbf{F}'\mathbf{1}} \tag{5.43}$$

My choice of a vector of 1's is quite arbitrary. I might as well have chosen a vector of random numbers. Bill Symes, who suggested this approach to me, suggests using an observed data vector $\mathbf{d}$ for the data space weight, and $\mathbf{F}'\mathbf{d}$ for the model space weight. This requires an additional step, dividing out the units of the data $\mathbf{d}$.

Experience tells me that a broader methodology than all above is needed. Appropriate scaling is required in both data space and model space. We need two other weights $\mathbf{W}_m$ and $\mathbf{W}_d$ where $\hat{\mathbf{m}} = \mathbf{W}_m\mathbf{F}'\mathbf{W}_d\mathbf{d}$.

I have a useful practical example (stacking in $v(z)$ media) in another of my electronic books (BEI), where I found both $\mathbf{W}_m$ and $\mathbf{W}_d$ by iterative guessing. First

assume $\mathbf{W}_d = \mathbf{I}$ and estimate $\mathbf{W}_m$ as above. Then assume you have the correct $\mathbf{W}_m$ and estimate $\mathbf{W}_d$ as above. Iterate. (Perhaps some theorist can find a noniterative solution.) I believe this iterative procedure leads us to the best diagonal pre- and post- multipliers for any operator $\mathbf{F}$. By this I mean that the modified operator $(\mathbf{W}_d\mathbf{F}\mathbf{W}_m)$ is as close to being unitary as we will be able to obtain with diagonal transformation. Unitary means it is energy conserving and that the inverse is simply the conjugate transpose.

What good is it that $(\mathbf{W}_d\mathbf{F}\mathbf{W}_m)'(\mathbf{W}_d\mathbf{F}\mathbf{W}_m) \approx \mathbf{I}$? It gives us the most rapid convergence of least squares problems of the form

$$\mathbf{0} \quad \approx \quad \mathbf{W}_d(\mathbf{F}\mathbf{m} - \mathbf{d}) \quad = \quad \mathbf{W}_d(\mathbf{F}\mathbf{W}_m\mathbf{p} - \mathbf{d}) \tag{5.44}$$

Thus it defines for us the best diagonal transform to a preconditioning variable $\mathbf{p} = \mathbf{W}_m^{-1}\mathbf{m}$ to use during iteration, and suggests to us what residual weighting function we need to use if rapid convergence is a high priority. Suppose we are not satisfied with $\mathbf{W}_d$ being the weighting function for residuals. Equation (5.44) could still help us speed iteration. Instead of beginning iteration with $\mathbf{p} = \mathbf{0}$, we could begin from the solution $\mathbf{p}$ to the regression (5.44).

The PhD thesis of James Rickett experiments extensively with data space and

model space weighting functions in the context of seismic velocity estimation.

5.9. A FORMAL DEFINITION FOR ADJOINTS

In mathematics, adjoints are defined a little differently than we have defined them here (as matrix transposes).[2] The mathematician begins by telling us that we cannot simply form any dot product we want. We are not allowed to take the dot product of any two vectors in model space $\mathbf{m}_1 \cdot \mathbf{m}_2$ or data space $\mathbf{d}_1 \cdot \mathbf{d}_2$. Instead, we must first transform them to a preferred coordinate system. Say $\tilde{\mathbf{m}}_1 = \mathbf{M}\mathbf{m}_1$ and $\tilde{\mathbf{d}}_1 = \mathbf{D}\mathbf{d}_1$, etc for other vectors. We complain we do not know $\mathbf{M}$ and $\mathbf{D}$. They reply that we do not really need to know them but we do need to have the inverses (aack!) of $\mathbf{M}'\mathbf{M}$ and $\mathbf{D}'\mathbf{D}$. A pre-existing common notation is $\sigma_m^{-2} = \mathbf{M}'\mathbf{M}$ and $\sigma_d^{-2} = \mathbf{D}'\mathbf{D}$. Now the mathematician buries the mysterious new positive-definite matrix inverses in the definition of dot product $< \mathbf{m}_1, \mathbf{m}_2 >= \mathbf{m}_1' \mathbf{M}'\mathbf{M}\mathbf{m}_2 = \mathbf{m}_1' \sigma_m^{-2} \mathbf{m}_2$ and likewise with $< \mathbf{d}_1, \mathbf{d}_2 >$. This suggests a total reorganization of our programs. Instead of computing $(\mathbf{m}_1'\mathbf{M}')(\mathbf{M}\mathbf{m}_2)$ we could compute $\mathbf{m}_1'(\sigma_m^{-2}\mathbf{m}_2)$. Indeed, this is the

[2] I would like to thank Albert Tarantola for suggesting this topic.

"conventional" approach. This definition of dot product would be buried in the solver code. The other thing that would change would be the search direction $\Delta\mathbf{m}$. Instead of being the gradient as we have defined it $\Delta\mathbf{m} = \mathbf{L}'\mathbf{r}$, it would be $\Delta\mathbf{m} = \sigma_m^{-2}\mathbf{L}'\sigma_d^{-2}\mathbf{r}$. A mathematician would *define* the adjoint of $\mathbf{L}$ to be $\sigma_m^{-2}\mathbf{L}'\sigma_d^{-2}$. (Here $\mathbf{L}'$ remains matrix transpose.) You might notice this approach nicely incorporates both residual weighting and preconditioning while yet evading the issue of where we get the matrices σ_m^2 and σ_d^2 or how we invert them. Fortunately, upcoming chapter 6 suggests how, in image estimation problems, to obtain sensible estimates of the elusive operators $\mathbf{M}$ and $\mathbf{D}$. Paranthetically, modeling calculations in physics and engineering often use similar mathematics in which the role of $\mathbf{M}'\mathbf{M}$ is not so mysterious. Kinetic energy is mass times velocity squared. Mass can play the role of $\mathbf{M}'\mathbf{M}$.

So, should we continue to use $(\mathbf{m}_1'\mathbf{M}')(\mathbf{M}\mathbf{m}_2)$ or should we take the conventional route and go with $\mathbf{m}_1'(\sigma_m^{-2}\mathbf{m}_2)$? One day while benchmarking a wide variety of computers I was shocked to see some widely differing numerical results. Now I know why. Consider adding 10^7 identical positive floating point numbers, say 1.0's, in an arithmetic with precision of 10^{-6}. After you have added in the first 10^6 numbers, the rest will all truncate in the roundoff and your sum will be wrong by

a factor of ten. If the numbers were added in pairs, and then the pairs added, etc, there would be no difficulty. Precision is scary stuff!

It is my understanding and belief that there is nothing wrong with the approach of this book, in fact, it seems to have some definite advantages. While the conventional approach requires one to compute the adjoint correctly, we do not. The method of this book (which I believe is properly called conjugate directions) has a robustness that, I'm told, has been superior in some important geophysical applications. The conventional approach seems to get in trouble when transpose operators are computed with insufficient precision.

Chapter 6

Multidimensional autoregression

The many applications of least squares to the one-dimensional convolution operator constitute the subject known as "time-series analysis." The *autoregression* filter, also known as the prediction-error filter (PEF), gathers statistics for us, not the au-

tocorrelation or the spectrum directly but it gathers them indirectly as the inverse of the amplitude spectrum of its input. The PEF plays the role of the so-called "inverse-covariance matrix" in statistical estimation theory. Given the PEF, we use it to find missing portions of signals.

6.0.1. Time domain versus frequency domain

In the simplest applications, solutions can be most easily found in the frequency domain. When complications arise, it is better to use the time domain, to directly apply the convolution operator and the method of least squares.

A first complicating factor in the frequency domain is a required boundary in the time domain, such as that between past and future, or requirements that a filter be nonzero in a stated time interval. Another factor that attracts us to the time domain rather than the frequency domain is weighting functions.

Weighting functions are appropriate whenever a signal or image amplitude varies from place to place. Much of the literature on time-series analysis applies to the limited case of uniform weighting functions. Such time series are said to be "stationary." This means that their statistical properties do not change in time. In

real life, particularly in the analysis of echos, signals are never stationary in time and space. A stationarity assumption is a reasonable starting assumption, but we should know how to go beyond it so we can take advantage of the many opportunities that do arise. In order of increasing difficulty in the frequency domain are the following complications:

1. A time boundary such as between past and future.
2. More time boundaries such as delimiting a filter.
3. More time boundaries such as erratic locations of missing data.
4. Nonstationary signal, i.e., time-variable weighting.
5. Time-axis stretching such as normal moveout.

We will not have difficulty with any of these complications here, because we will stay in the time domain and set up and solve optimization problems by use of the conjugate-direction method. Thus we will be able to cope with great complexity in goal formulation and get the right answer without approximations. By contrast, analytic or partly analytic methods can be more economical, but they generally solve somewhat different problems than those given to us by nature.

6.1. SOURCE WAVEFORM, MULTIPLE REFLECTIONS

Here we devise a simple mathematical model for deep water bottom multiple reflections.[1] There are two unknown waveforms, the source waveform $S(\omega)$ and the ocean-floor reflection $F(\omega)$. The water-bottom primary reflection $P(\omega)$ is the convolution of the source waveform with the water-bottom response; so $P(\omega) = S(\omega)F(\omega)$. The first multiple reflection $M(\omega)$ sees the same source waveform, the ocean floor, a minus one for the free surface, and the ocean floor again. Thus the observations $P(\omega)$ and $M(\omega)$ as functions of the physical parameters are

$$P(\omega) = S(\omega)\,F(\omega) \tag{6.1}$$

$$M(\omega) = -S(\omega)\,F(\omega)^2 \tag{6.2}$$

[1] For this short course I am omitting here many interesting examples of multiple reflections shown in my 1992 book, PVI.

Algebraically the solutions of equations (6.1) and (6.2) are

$$F(\omega) = -M(\omega)/P(\omega) \tag{6.3}$$

$$S(\omega) = -P(\omega)^2/M(\omega) \tag{6.4}$$

These solutions can be computed in the Fourier domain by simple division. The difficulty is that the divisors in equations (6.3) and (6.4) can be zero, or small. This difficulty can be attacked by use of a positive number ϵ to stabilize it. For example, multiply equation (6.3) on top and bottom by $P'(\omega)$ and add $\epsilon > 0$ to the denominator. This gives

$$F(\omega) = -\frac{M(\omega)P'(\omega)}{P(\omega)P'(\omega)+\epsilon} \tag{6.5}$$

where $P'(\omega)$ is the complex conjugate of $P(\omega)$. Although the ϵ stabilization seems nice, it apparently produces a nonphysical model. For ϵ large or small, the time-domain response could turn out to be of much greater duration than is physically reasonable. This should not happen with perfect data, but in real life, data always has a limited spectral band of good quality.

Functions that are rough in the frequency domain will be long in the time do-

main. This suggests making a short function in the time domain by local smoothing in the frequency domain. Let the notation $< \cdots >$ denote smoothing by local averaging. Thus, to specify filters whose time duration is not unreasonably long, we can revise equation (6.5) to

$$F(\omega) \quad = \quad - \frac{< M(\omega) P'(\omega) >}{< P(\omega) P'(\omega) >} \tag{6.6}$$

where instead of deciding a size for ϵ we need to decide how much smoothing. I find that smoothing has a simpler physical interpretation than choosing ϵ. The goal of finding the filters $F(\omega)$ and $S(\omega)$ is to best model the multiple reflections so that they can be subtracted from the data, and thus enable us to see what primary reflections have been hidden by the multiples.

These frequency-duration difficulties do not arise in a time-domain formulation. Unlike in the frequency domain, in the time domain it is easy and natural to limit the duration and location of the nonzero time range of $F(\omega)$ and $S(\omega)$. First express (6.3) as

$$0 \quad = \quad P(\omega) F(\omega) + M(\omega) \tag{6.7}$$

To imagine equation (6.7) as a fitting goal in the time domain, instead of scalar

functions of ω, think of vectors with components as a function of time. Thus **f** is a column vector containing the unknown sea-floor filter, **m** contains the "multiple" portion of a seismogram, and **P** is a matrix of down-shifted columns, each column being the "primary".

$$\mathbf{0} \quad \approx \quad \mathbf{r} \quad = \quad \begin{bmatrix} r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \\ r_6 \\ r_7 \\ r_8 \end{bmatrix} \quad = \quad \begin{bmatrix} p_1 & 0 & 0 \\ p_2 & p_1 & 0 \\ p_3 & p_2 & p_1 \\ p_4 & p_3 & p_2 \\ p_5 & p_4 & p_3 \\ p_6 & p_5 & p_4 \\ 0 & p_6 & p_5 \\ 0 & 0 & p_6 \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \\ f_3 \end{bmatrix} + \begin{bmatrix} m_1 \\ m_2 \\ m_3 \\ m_4 \\ m_5 \\ m_6 \\ m_7 \\ m_8 \end{bmatrix} \tag{6.8}$$

6.2. TIME-SERIES AUTOREGRESSION

Given y_t and y_{t-1}, you might like to predict y_{t+1}. The prediction could be a scaled sum or difference of y_t and y_{t-1}. This is called "autoregression" because a signal

is regressed on itself. To find the scale factors you would optimize the fitting goal below, for the prediction filter (f_1, f_2):

$$\mathbf{0} \quad \approx \quad \mathbf{r} \quad = \quad \begin{bmatrix} y_1 & y_0 \\ y_2 & y_1 \\ y_3 & y_2 \\ y_4 & y_3 \\ y_5 & y_4 \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \end{bmatrix} - \begin{bmatrix} y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix} \tag{6.9}$$

(In practice, of course the system of equations would be much taller, and perhaps somewhat wider.) A typical row in the matrix (6.9) says that $y_{t+1} \approx y_t\, f_1 + y_{t-1}\, f_2$ hence the description of f as a "prediction" filter. The error in the prediction is simply the residual. Define the residual to have opposite polarity and merge the column vector into the matrix, so you get

$$\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad \approx \quad \mathbf{r} \quad = \quad \begin{bmatrix} y_2 & y_1 & y_0 \\ y_3 & y_2 & y_1 \\ y_4 & y_3 & y_2 \\ y_5 & y_4 & y_3 \\ y_6 & y_5 & y_4 \end{bmatrix} \begin{bmatrix} 1 \\ -f_1 \\ -f_2 \end{bmatrix} \tag{6.10}$$

which is a standard form for autoregressions and prediction error.

Multiple reflections are predictable. It is the unpredictable part of a signal, the prediction residual, that contains the primary information. The output of the filter $(1, -f_1, -f_2) = (a_0, a_1, a_2)$ is the unpredictable part of the input. This filter is a simple example of a "prediction-error" (PE) filter. It is one member of a family of filters called "error filters."

The error-filter family are filters with one coefficient constrained to be unity and various other coefficients constrained to be zero. Otherwise, the filter coefficients are chosen to have minimum power output. Names for various error filters follow:

$(1, a_1, a_2, a_3, \cdots, a_n)$	**prediction-error (PE) filter**
$(1, 0, 0, a_3, a_4, \cdots, a_n)$	gapped PE filter with a gap
$(a_{-m}, \cdots, a_{-2}, a_{-1}, 1, a_1, a_2, a_3, \cdots, a_n)$	**interpolation-error (IE) filter**

We introduce a free-mask matrix **K** which "passes" the freely variable coefficients in the filter and "rejects" the constrained coefficients (which in this first

example is merely the first coefficient $a_0 = 1$).

$$\mathbf{K} \quad = \quad \begin{bmatrix} 0 & . & . \\ . & 1 & . \\ . & . & 1 \end{bmatrix} \tag{6.11}$$

To compute a simple prediction error filter $\mathbf{a} = (1, a_1, a_2)$ with the CD method, we write (6.9) or (6.10) as

$$\mathbf{0} \quad \approx \quad \mathbf{r} \quad = \quad \begin{bmatrix} y_2 & y_1 & y_0 \\ y_3 & y_2 & y_1 \\ y_4 & y_3 & y_2 \\ y_5 & y_4 & y_3 \\ y_6 & y_5 & y_4 \end{bmatrix} \begin{bmatrix} 0 & . & . \\ . & 1 & . \\ . & . & 1 \end{bmatrix} \begin{bmatrix} 1 \\ a_1 \\ a_2 \end{bmatrix} + \begin{bmatrix} y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix} \tag{6.12}$$

Let us move from this specific fitting goal to the general case. (Notice the similarity of the free-mask matrix **K** in this filter estimation problem with the free-mask matrix

J in missing data goal (3.3).) The fitting goal is,

$$\mathbf{0} \approx \mathbf{Ya} \tag{6.13}$$

$$\mathbf{0} \approx \mathbf{Y}(\mathbf{I}-\mathbf{K}+\mathbf{K})\mathbf{a} \tag{6.14}$$

$$\mathbf{0} \approx \mathbf{YKa}+\mathbf{Y}(\mathbf{I}-\mathbf{K})\mathbf{a} \tag{6.15}$$

$$\mathbf{0} \approx \mathbf{YKa}+\mathbf{Ya}_0 \tag{6.16}$$

$$\mathbf{0} \approx \mathbf{YKa}+\mathbf{y} \tag{6.17}$$

$$\mathbf{0} \approx \mathbf{r} = \mathbf{YKa}+\mathbf{r}_0 \tag{6.18}$$

which means we initialize the residual with $\mathbf{r}_0 = \mathbf{y}$. and then iterate with

$$\Delta\mathbf{a} \longleftarrow \mathbf{K}'\mathbf{Y}'\,\mathbf{r} \tag{6.19}$$

$$\Delta\mathbf{r} \longleftarrow \mathbf{YK}\,\Delta\mathbf{a} \tag{6.20}$$

6.3. PREDICTION-ERROR FILTER OUTPUT IS WHITE

- **The relationship between spectrum and PEF**

Knowledge of an autocorrelation function is equivalent to knowledge of a spectrum. The two are simply related by Fourier transform. A spectrum or an autocorrelation function encapsulates an important characteristic of a signal or an image. Generally the spectrum changes slowly from place to place although it could change rapidly. Of all the assumptions we could make to fill empty bins, one that people usually find easiest to agree with is that the spectrum should be the same in the empty-bin regions as where bins are filled. In practice we deal with neither the spectrum nor its autocorrelation but with a third object. This third object is the Prediction Error Filter (PEF), the filter in equation (6.10).

Take equation (6.10) for $\mathbf{r}$ and multiply it by the adjoint $\mathbf{r}'$ getting a quadratic form in the PEF coefficients. Minimizing this quadratic form determines the PEF. This quadratic form depends only on the autocorrelation of the original data y_t, not on the data y_t itself. Clearly the PEF is unchanged if the data has its polarity reversed or its time axis reversed. Indeed, we'll see here that knowledge of the PEF

is equivalent to knowledge of the autocorrelation or the spectrum.

- **Undoing convolution in nature**

Prediction-error filtering is also called "deconvolution". This word goes back to very basic models and concepts. In this model one envisions a random white-spectrum excitation function **x** existing in nature, and this excitation function is somehow filtered by unknown natural processes, with a filter operator **B** producing an *output* **y** in nature that becomes the *input* **y** to our computer programs. This is sketched in Figure 6.1. Then we design a prediction-error filter **A** on **y**, which yields

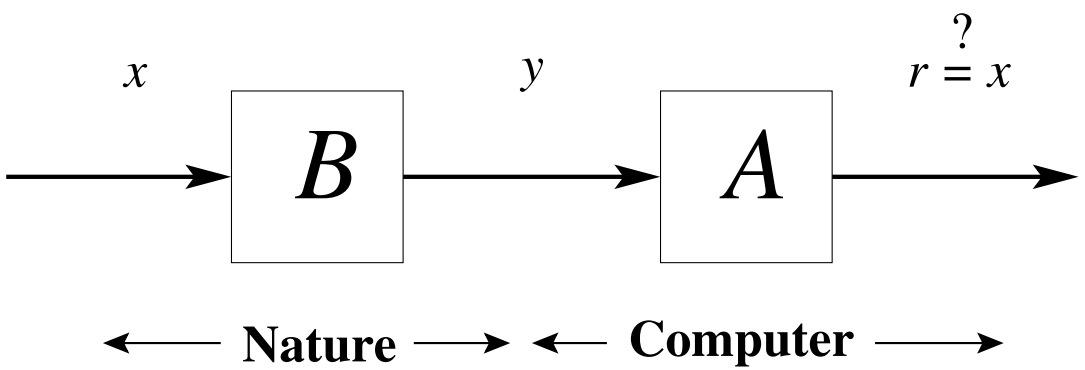


Figure 6.1: Flow of information from nature, to observation, into computer. mda-systems [NR]

a white-spectrumed output residual **r**. Because **r** and **x** theoretically have the same

spectrum, the tantalizing prospect is that maybe **r** equals **x**, meaning that the PEF **A** has *deconvolved* the unknown convolution **B**.

- **Causal with causal inverse**

Theoretically, a PEF is a causal filter with a causal inverse. This adds confidence to the likelihood that deconvolution of natural processes with a PEF might get the correct phase spectrum as well as the correct amplitude spectrum. Naturally, the PEF does not give the correct phase to an "all-pass" filter. That is a filter with a phase shift but a constant amplitude spectrum. (I think most migration operators are in this category.)

Theoretically we should be able to use a PEF in either convolution or polynomial division. There are some dangers though, mainly connected with dealing with data in small windows. Truncation phenomena might give us PEF estimates that are causal, but whose inverse is not, so they cannot be used in polynomial division. This is a lengthy topic in the classic literature. This old, fascinating subject is examined in my books, FGDP and PVI. A classic solution is one by John Parker Burg. We should revisit the Burg method in light of the helix.

- **PEF output tends to whiteness**

The most important property of a prediction-error filter or PEF is that its output tends to a white spectrum (to be proven here). No matter what the input to this filter, its output tends to whiteness as the number of the coefficients $n \to \infty$ tends to infinity. Thus, the PE filter adapts itself to the input by absorbing all its color. This has important statistical implications and important geophysical implications.

- **Spectral estimation**

The PEF's output being white leads to an important consequence: To specify a spectrum, we can give the spectrum (of an input) itself, give its autocorrelation, or give its PEF coefficients. Each is transformable to the other two. Indeed, an effective mechanism of spectral estimation, developed by John P. Burg and described in FGDP, is to compute a PE filter and look at the inverse of its spectrum.

- **Short windows**

The power of a PE filter is that a short filter can often extinguish, and thereby represent, the information in a long resonant filter. If the input to the PE filter is a sinusoid, it is exactly predictable by a three-term recurrence relation, and all the

color is absorbed by a three-term PE filter (see exercises). Burg's spectral estimation is especially effective in short windows.

- **Weathered layer resonance**

That the output spectrum of a PE filter is white is also useful geophysically. Imagine the reverberation of the soil layer, highly variable from place to place, as the resonance between the surface and shallow more-consolidated soil layers varies rapidly with surface location because of geologically recent fluvial activity. The spectral color of this erratic variation on surface-recorded seismograms is compensated for by a PE filter. Usually we do not want PE-filtered seismograms to be white, but once they all have the same spectrum, it is easy to postfilter them to any desired spectrum.

6.3.1. PEF whiteness proof in 1-D

The basic idea of least-squares fitting is that the residual is orthogonal to the fitting functions. Applied to the PE filter, this idea means that the output of a PE filter is orthogonal to lagged inputs. The orthogonality applies only for lags in the past,

because prediction knows only the past while it aims to the future. What we want to show here is different, namely, that the output is uncorrelated with *itself* (as opposed to the input) for lags in *both* directions; hence the output spectrum is white.

In (6.21) are two separate and independent autoregressions, $\mathbf{0} \approx \mathbf{Y}_a \mathbf{a}$ for finding the filter $\mathbf{a}$, and $\mathbf{0} \approx \mathbf{Y}_b \mathbf{b}$ for finding the filter $\mathbf{b}$. By noticing that the two matrices are really the same (except a row of zeros on the bottom of $\mathbf{Y}_a$ is a row in the top of $\mathbf{Y}_b$) we realize that the two regressions must result in the same filters $\mathbf{a} = \mathbf{b}$, and the residual $\mathbf{r}_b$ is a shifted version of $\mathbf{r}_a$. In practice, I visualize the matrix being a

thousand components tall (or a million) and a hundred components wide.

$$\mathbf{0} \approx \mathbf{r}_a = \begin{bmatrix} y_1 & 0 & 0 \\ y_2 & y_1 & 0 \\ y_3 & y_2 & y_1 \\ y_4 & y_3 & y_2 \\ y_5 & y_4 & y_3 \\ y_6 & y_5 & y_4 \\ 0 & y_6 & y_5 \\ 0 & 0 & y_6 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ a_1 \\ a_2 \end{bmatrix}; \qquad \mathbf{0} \approx \mathbf{r}_b = \begin{bmatrix} 0 & 0 & 0 \\ y_1 & 0 & 0 \\ y_2 & y_1 & 0 \\ y_3 & y_2 & y_1 \\ y_4 & y_3 & y_2 \\ y_5 & y_4 & y_3 \\ y_6 & y_5 & y_4 \\ 0 & y_6 & y_5 \\ 0 & 0 & y_6 \end{bmatrix} \begin{bmatrix} 1 \\ b_1 \\ b_2 \end{bmatrix} \tag{6.21}$$

When the energy $\mathbf{r}'\mathbf{r}$ of a residual has been minimized, the residual $\mathbf{r}$ is orthogonal to the fitting functions. For example, choosing a_2 to minimize $\mathbf{r}'\mathbf{r}$ gives $0 = \partial \mathbf{r}'\mathbf{r}/\partial a_2 = 2\mathbf{r}'\partial \mathbf{r}/\partial a_2$. This shows that $\mathbf{r}'$ is perpendicular to $\partial \mathbf{r}/\partial a_2$ which is the rightmost column of the $\mathbf{Y}_a$ matrix. Thus the vector $\mathbf{r}_a$ is orthogonal to all the columns in the $\mathbf{Y}_a$ matrix except the first (because we do not minimize with respect to a_0).

Our goal is a different theorem that is imprecise when applied to the three coefficient filters displayed in (6.21), but becomes valid as the filter length tends to infinity $\mathbf{a} = (1, a_1, a_2, a_3, \cdots)$ and the matrices become infinitely wide. Actually, all we require is the last component in $\mathbf{b}$, namely b_n tend to zero. This generally happens because as n increases, y_{t-n} becomes a weaker and weaker predictor of y_t.

The matrix $\mathbf{Y}_a$ contains all of the columns that are found in $\mathbf{Y}_b$ except the last (and the last one is not important). This means that $\mathbf{r}_a$ is not only orthogonal to all of $\mathbf{Y}_a$'s columns (except the first) but $\mathbf{r}_a$ is also orthogonal to all of $\mathbf{Y}_b$'s columns except the last. Although $\mathbf{r}_a$ isn't really perpendicular to the last column of $\mathbf{Y}_b$, it doesn't matter because that column has hardly any contribution to $\mathbf{r}_b$ since $|b_n| << 1$. Because $\mathbf{r}_a$ is (effectively) orthogonal to all the components of $\mathbf{r}_b$, $\mathbf{r}_a$ is also orthogonal to $\mathbf{r}_b$ itself. (For any $\mathbf{u}$ and $\mathbf{v}$, if $\mathbf{r} \cdot \mathbf{u} = 0$ and $\mathbf{r} \cdot \mathbf{v} = 0$ then $\mathbf{r} \cdot (\mathbf{u} + \mathbf{v}) = 0$ and also $\mathbf{r} \cdot (a_1\mathbf{u} + a_2\mathbf{v}) = 0$).

Here is a detail: In choosing the example of equation (6.21), I have shifted the two fitting problems by only one lag. We would like to shift by more lags and get the same result. For this we need more filter coefficients. By adding many more filter coefficients we are adding many more columns to the right side of $\mathbf{Y}_b$. That's good because we'll be needing to neglect more columns as we shift $\mathbf{r}_b$ further from $\mathbf{r}_a$.

Neglecting these columns is commonly justified by the experience that "after short range regressors have had their effect, long range regressors generally find little remaining to predict." (Recall that the damped harmonic oscillator from physics, the finite difference equation that predicts the future from the past, uses only two lags.)

Here is the main point: Since $\mathbf{r}_b$ and $\mathbf{r}_a$ both contain the same signal $\mathbf{r}$ but time-shifted, the orthogonality at all shifts means that the autocorrelation of $\mathbf{r}$ vanishes at all lags. An exception, of course, is at zero lag. The autocorrelation does not vanish there because $\mathbf{r}_a$ is not orthogonal to its first column (because we did not minimize with respect to a_0).

As we redraw $\mathbf{0} \approx \mathbf{r}_b = \mathbf{Y}_b\mathbf{b}$ for various lags, we may shift the columns only downward because shifting them upward would bring in the first column of $\mathbf{Y}_a$ and the residual $\mathbf{r}_a$ is not orthogonal to that. Thus we have only proven that one side of the autocorrelation of $\mathbf{r}$ vanishes. That is enough however, because autocorrelation functions are symmetric, so if one side vanishes, the other must also.

If $\mathbf{a}$ and $\mathbf{b}$ were two-sided filters like $(\cdots, b_{-2}, b_{-1}, 1, b_1, b_2, \cdots)$ the proof would break. If $\mathbf{b}$ were two-sided, $\mathbf{Y}_b$ would catch the nonorthogonal column of $\mathbf{Y}_a$. Not only is $\mathbf{r}_a$ not proven to be perpendicular to the first column of $\mathbf{Y}_a$, but it cannot be

orthogonal to it because a signal cannot be orthogonal to itself.

The implications of this theorem are far reaching. The residual **r**, a convolution of **y** with **a** has an autocorrelation that is an impulse function. The Fourier transform of an impulse is a constant. Thus the spectrum of the residual is "white". Thus **y** and **a** have mutually inverse spectra.

> Since the output of a PEF is white, the PEF itself has a spectrum inverse to its input.

An important application of the PEF is in missing data interpolation. We'll see examples later in this chapter. My third book, PVI[2] has many examples[3] in one dimension with both synthetic data and field data including the gap parameter. Here we next extend these ideas to two (or more) dimensions.

---

[2] http://sepwww.stanford.edu/sep/prof/pvi/toc_html/index.html

[3] http://sepwww.stanford.edu/sep/prof/pvi/tsa/paper_html/node1.html

6.3.2. Simple dip filters

Convolution in two dimensions is just like convolution in one dimension except that convolution is done on two axes. The input and output data are planes of numbers and the filter is also a plane. A two-dimensional filter is a small plane of numbers that is convolved over a big data plane of numbers.

Suppose the data set is a collection of seismograms uniformly sampled in space. In other words, the data is numbers in a (t,x)-plane. For example, the following filter destroys any wavefront aligned along the direction of a line containing both the "+1" and the "-1".

$$\begin{matrix} -1 & \cdot \\ \cdot & \cdot \\ \cdot & 1 \end{matrix} \tag{6.22}$$

The next filter destroys a wave with a slope in the opposite direction:

$$\begin{matrix} \cdot & 1 \\ -1 & \cdot \end{matrix} \tag{6.23}$$

To convolve the above two filters, we can reverse either on (on both axes) and cor-

relate them, so that you can get

$$\begin{matrix} . & -1 & . \\ 1 & . & . \\ . & . & 1 \\ . & -1 & . \end{matrix} \tag{6.24}$$

which destroys waves of both slopes.

A two-dimensional filter that can be a dip-rejection filter like (6.22) or (6.23) is

$$\begin{matrix} a & . \\ b & . \\ c & 1 \\ d & . \\ e & . \end{matrix} \tag{6.25}$$

where the coefficients (a,b,c,d,e) are to be estimated by least squares in order to minimize the power out of the filter. (In the filter table, the time axis runs vertically.)

Fitting the filter to two neighboring traces that are identical but for a time shift, we see that the filter coefficients (a,b,c,d,e) should turn out to be some-

thing like $(-1,0,0,0,0)$ or $(0,0,-.5,-.5,0)$, depending on the dip (stepout) of the data. But if the two channels are not fully coherent, we expect to see something like $(-.9,0,0,0,0)$ or $(0,0,-.4,-.4,0)$. To find filters such as (6.24), we adjust coefficients to minimize the power out of filter shapes, as in

$$\begin{array}{ccc} v & a & \cdot \\ w & b & \cdot \\ x & c & 1 \\ y & d & \cdot \\ z & e & \cdot \end{array} \tag{6.26}$$

With 1-dimensional filters, we think mainly of power spectra, and with 2-dimensional filters we can think of temporal spectra and spatial spectra. What is new, however, is that in two dimensions we can think of dip spectra (which is when a 2-dimensional spectrum has a particularly common form, namely when energy organizes on radial lines in the (ω, k_x)-plane). As a short (three-term) 1-dimensional filter can devour a sinusoid, we have seen that simple 2-dimensional filters can devour a small number of dips.

6.3.3. PEF whiteness proof in 2-D

A well-known property (see FGDP or PVI) of a 1-D PEF is that its energy clusters immediately after the impulse at zero delay time. Applying this idea to the helix in Figure 4.2 hlx/fig:sergey-helix shows us that we can consider a 2-D PEF to be a small halfplane like (4.7) hlx/eqn:2dpef with an impulse along a side. These shapes are what we see here in Figure 6.2.

Figure 6.2 shows the input plane with a 2-D filter on top of it at two possible locations. The filter shape is a semidisk, which you should imagine being of infinitely large radius. Notice that semidisk A includes all the points in B. The output of disk A will be shown to be orthogonal to the output of disk B. Conventional least squares theory says that the coefficients of the filter are designed so that the output of the filter is orthogonal to each of the inputs to that filter (except for the input under the "1," because any nonzero signal cannot be orthogonal to itself). Recall that if a given signal is orthogonal to each in a given group of signals, then the given signal is orthogonal to all linear combinations within that group. The output at B is a linear combination of members of its input group, which is included in the input group of A, which are already orthogonal to A. Therefore the output at B is orthogonal to the

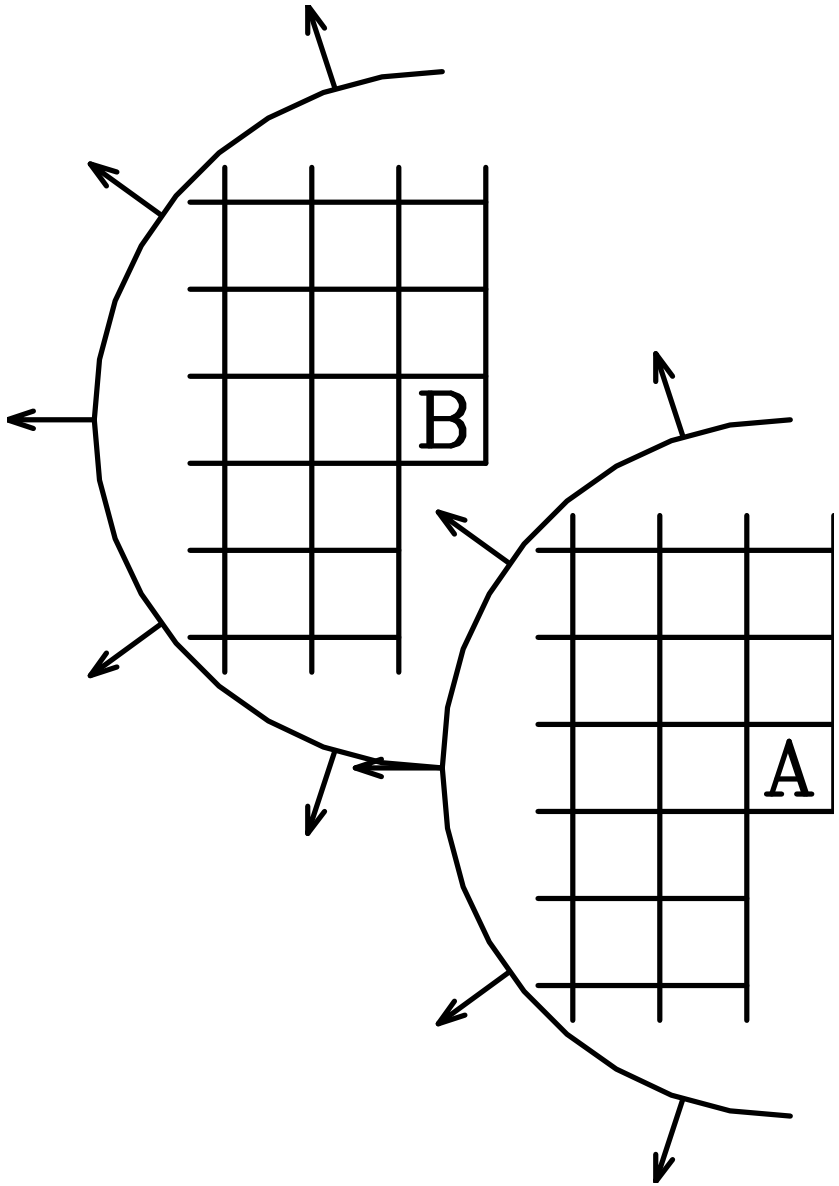


Figure 6.2: A 2-D whitening filter template, and itself lagged. At output locations "A" and "B," the filter coefficient is constrained to be "1". When the semicircles are viewed as having infinite radius, the B filter is contained in the A filter. Because the output at A is orthogonal to all its inputs, which include all inputs of B, the output at A is orthogonal to the output of B. mda-whitepruf [ER]

output at A. In summary,

residual	$\perp$	fitting function
output at A	$\perp$	each input to A
output at A	$\perp$	each input to B
output at A	$\perp$	linear combination of each input to B
output at A	$\perp$	output at B

The essential meaning is that a particular lag of the output autocorrelation function vanishes.

Study Figure 6.2 to see for what lags all the elements of the B filter are wholly contained in the A filter. These are the lags where we have shown the output autocorrelation to be vanishing. Notice another set of lags where we have proven nothing (where B is moved to the right of A). Autocorrelations are centrosymmetric, which means that the value at any lag is the same as the value at the negative of that lag, even in 2-D and 3-D where the lag is a vector quantity. Above we have shown that a halfplane of autocorrelation values vanishes. By the centrosymmetry, the other half must vanish too. Thus the autocorrelation of the PEF output is an impulse function, so its 2-D spectrum is white.

The helix tells us why the proper filter form is not a square with the “1” on the

corner. Before I discovered the helix, I understood it another way (that I learned from John P. Burg): For a spectrum to be white, *all* nonzero autocorrelation lags must be zero-valued. If the filter were a quarter-plane, then the symmetry of autocorrelations would only give us vanishing in another quarter, so there would be two remaining quarter-planes where the autocorrelation was not zero.

Fundamentally, the white-output theorem requires a one-dimensional ordering to the values in a plane or volume. The filter must contain a halfplane of values so that symmetry gives the other half.

You will notice some nonuniqueness. We could embed the helix with a 90° rotation in the original physical application. Besides the difference in side boundaries, the 2-D PEF would have a different orientation. Both PEFs should have an output that tends to whiteness as the filter is enlarged. It seems that we could design whitening autoregression filters for 45° rotations also, and we could also design them for hexagonal coordinate systems. In some physical problems, you might find the nonuniqueness unsettling. Does it mean the "final solution" is nonunique? Usually not, or not seriously so. Recall even in one dimension, the time reverse of a PEF has the same spectrum as the original PEF. When a PEF is used for regularizing a fitting problem, it is worth noticing that the quadratic form minimized is the PEF

times its adjoint so the phase drops out. Likewise, a missing data restoration also amounts to minimizing a quadratic form so the phase again drops out.

6.3.4. Examples of modeling and deconvolving with a 2-D PEF

Here we examine elementary signal-processing applications of 2-D prediction-error filters (PEFs) on both everyday 2-D textures and on seismic data. Some of these textures are easily modeled with prediction-error filters (PEFs) while others are not. All figures used the same 10×10 filter shape. No attempt was made to optimize filter size or shape or any other parameters.

Results in Figures 6.3-6.9 are shown with various familiar textures[4] on the left as training data sets. From these training data sets, a prediction-error filter (PEF) is estimated using module `pef` /prog:pef. The center frame is simulated data made by deconvolving (polynomial division) random numbers by the estimated PEF. The

[4] I thank Morgan Brown for finding these textures.

right frame is the more familiar process, convolving the estimated PEF on the training data set.

Theoretically, the right frame tends towards a white spectrum. Earlier you could notice the filter size by knowing that the output was taken to be zero where the filter is only partially on the data. This was annoying on real data where we didn't want to throw away any data around the sides. Now the filtering is done without a call to the boundary module so we have typical helix wraparound.

Since a PEF tends to the inverse of the spectrum of its input, results similar to these could probably be found using Fourier transforms, smoothing spectra, etc. We used PEFs because of their flexibility. The filters can be any shape. They can dodge around missing data, or we can use them to estimate missing data. We avoid periodic boundary assumptions inherent to FT. The PEF's are designed only internal to known data, not off edges so they are readily adaptible to nonstationarity. Thinking of these textures as seismic time slices, the textures could easily be required to pass thru specific values at well locations.

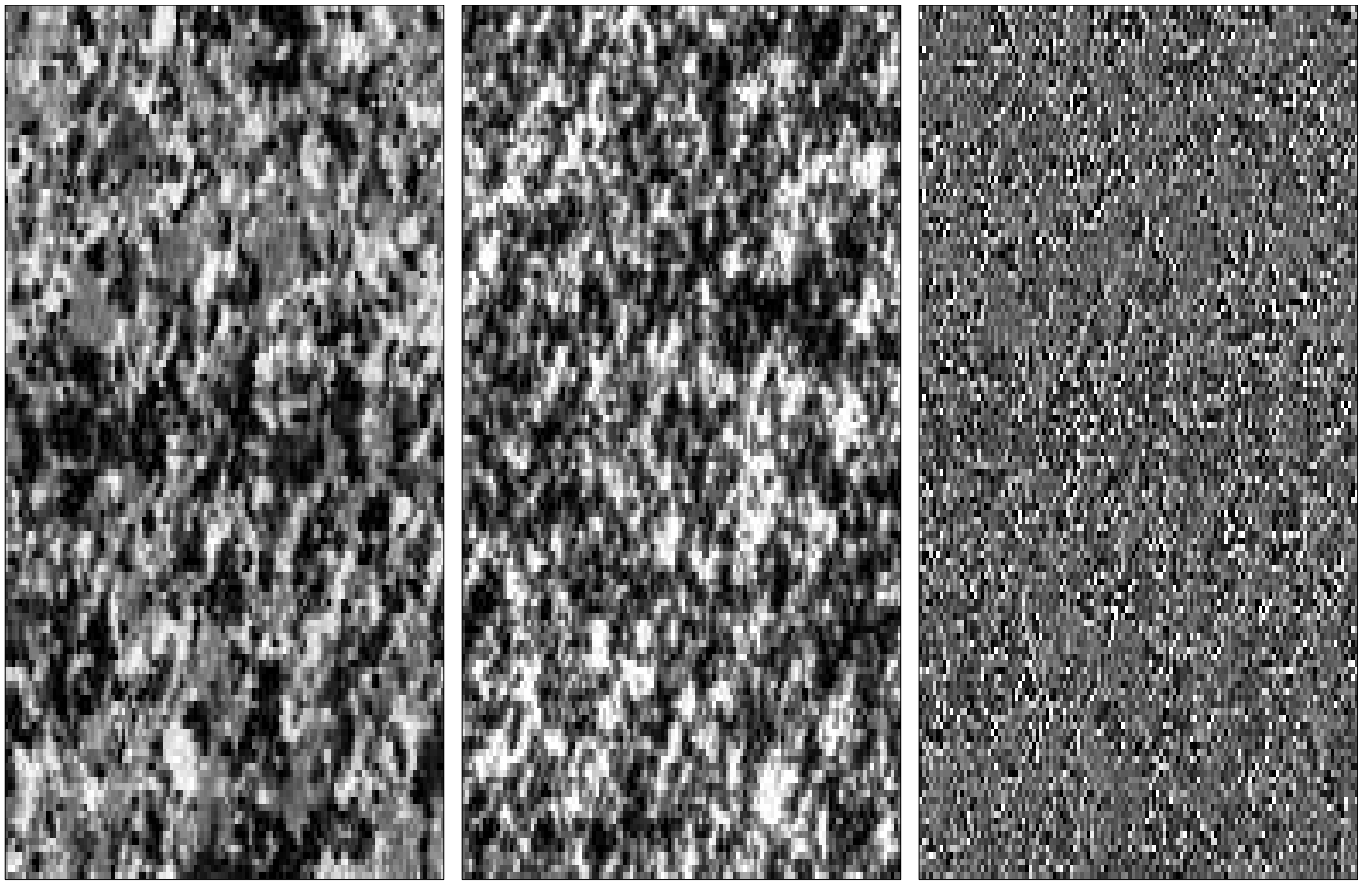


Figure 6.3: Synthetic granite matches the training image quite well. The prediction error (PE) is large at grain boundaries so it almost seems to outline the grains. mda-granite [ER]

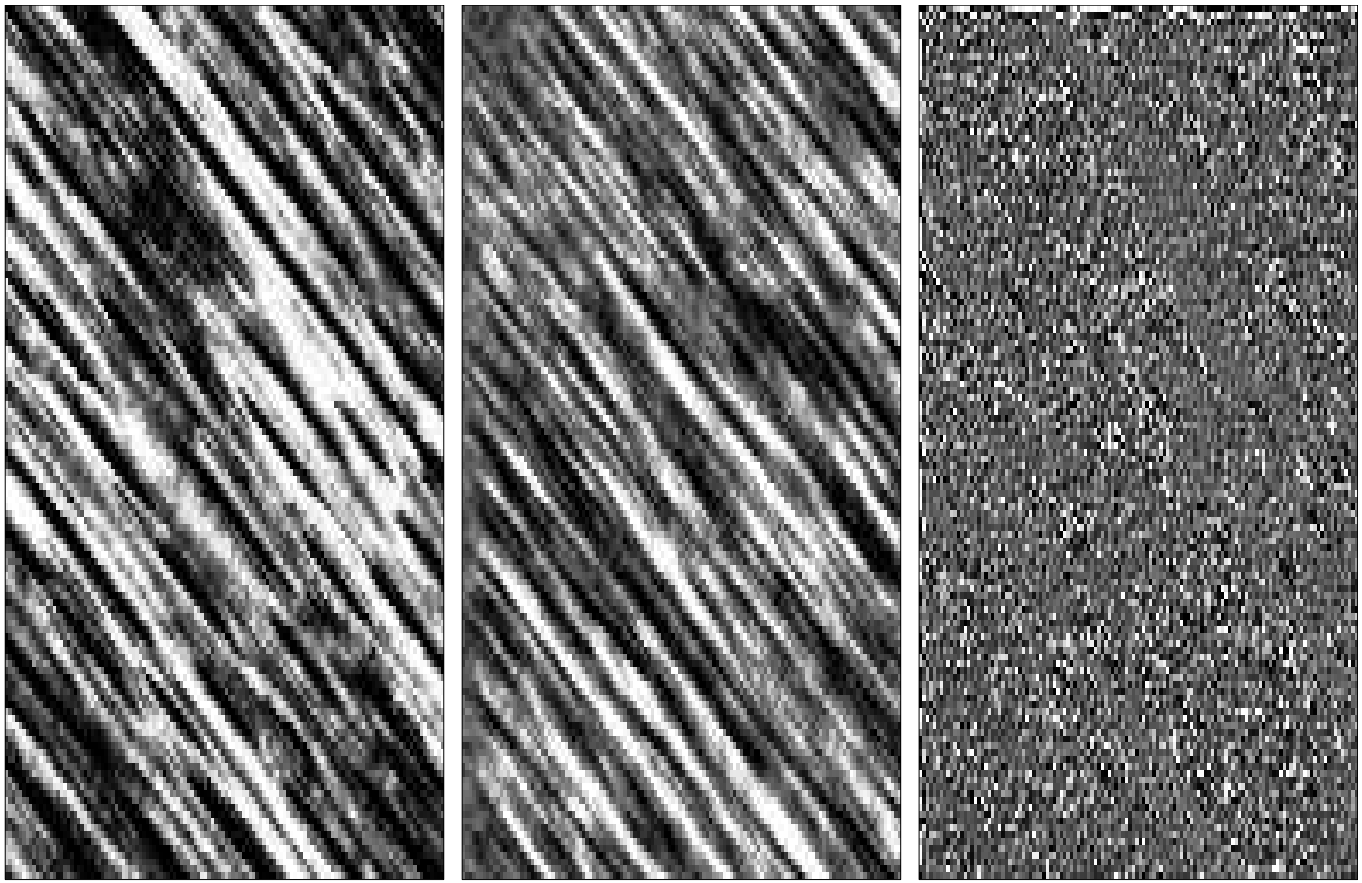


Figure 6.4: Synthetic wood grain has too little white. This is because of the non-symmetric brightness histogram of natural wood. Again, the PEF output looks random as expected. mda-wood [ER]

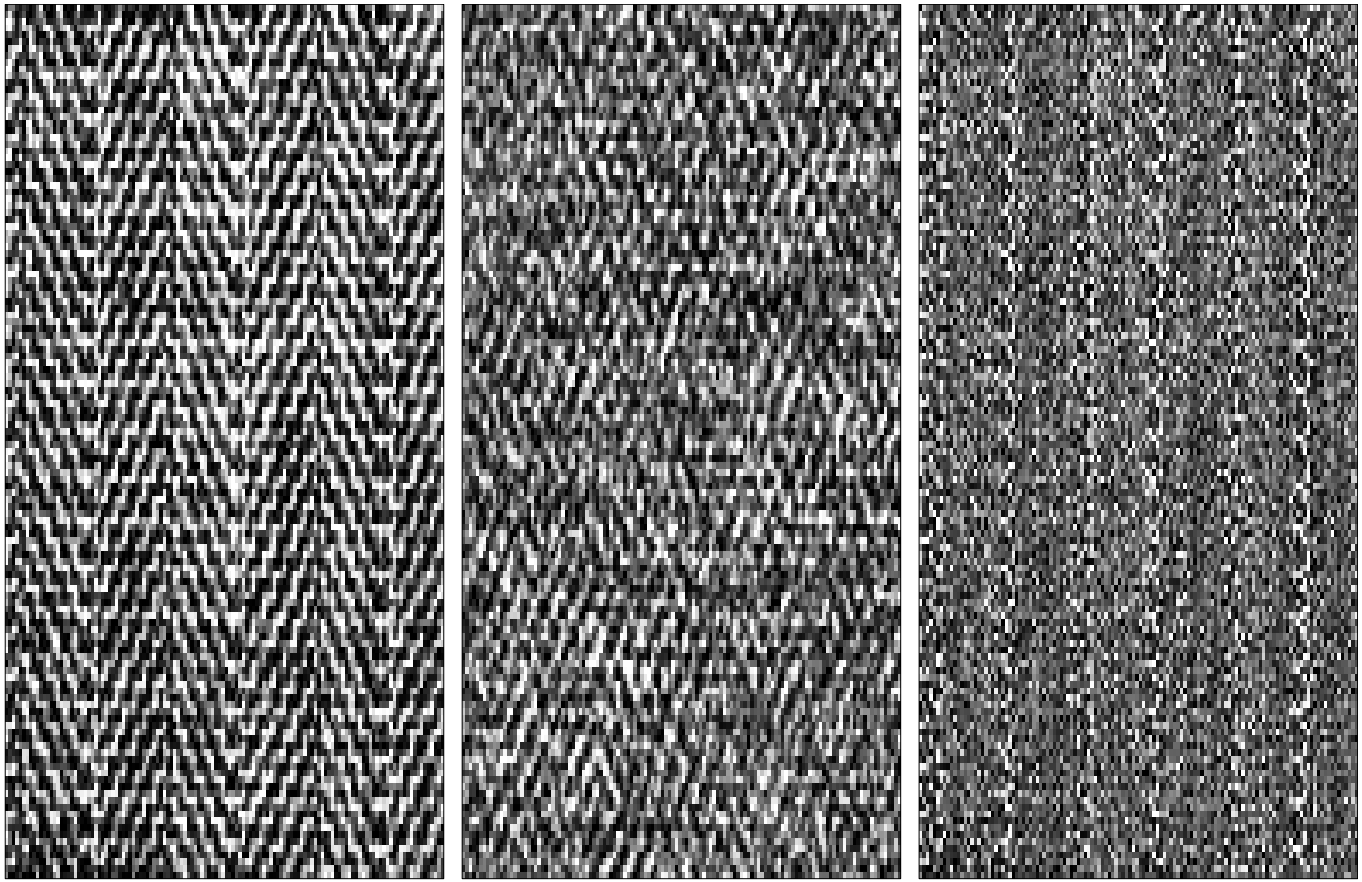


Figure 6.5: A banker's suit (left). A student's suit (center). My suit (right). The prediction error is large where the weave changes direction. mda-herr [ER]

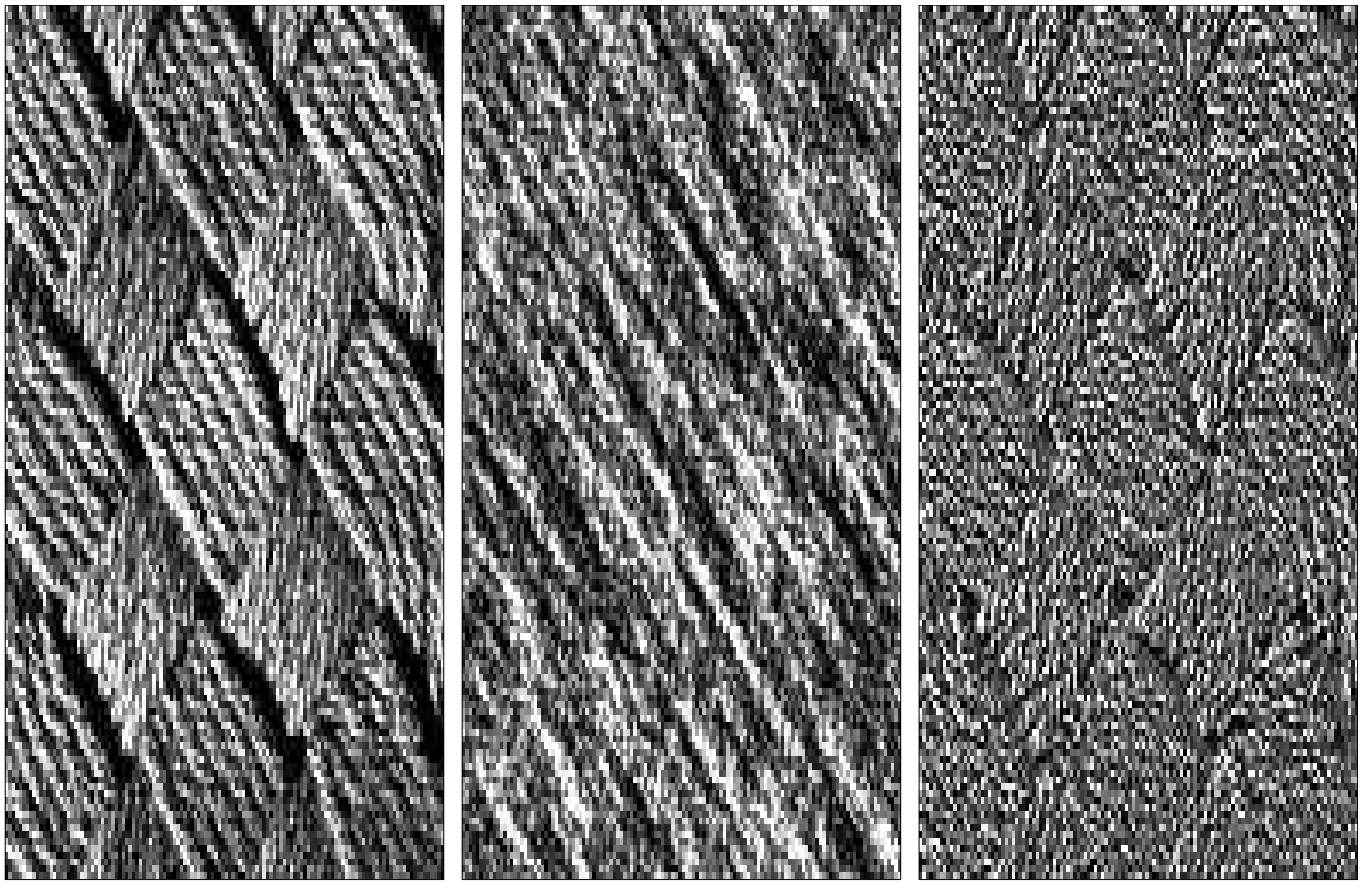


Figure 6.6: Basket weave. The simulated data fails to segregate the two dips into a checkerboard pattern. The PEF output looks structured perhaps because the filter is too small. mda-basket [ER]

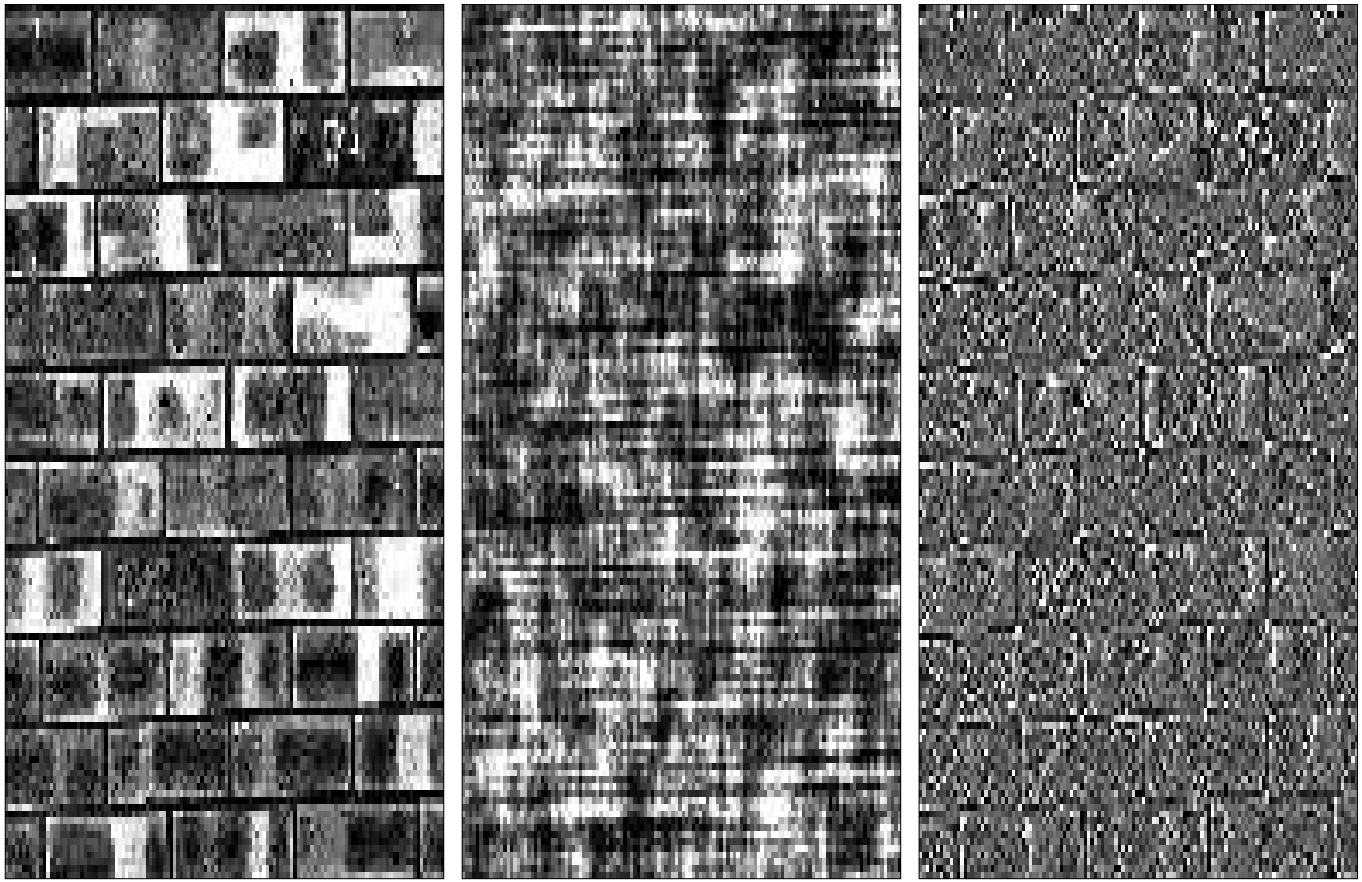


Figure 6.7: Brick. Synthetic brick edges are everywhere and do not enclose blocks containing a fixed color. PEF output highlights the mortar. mda-brick [ER]

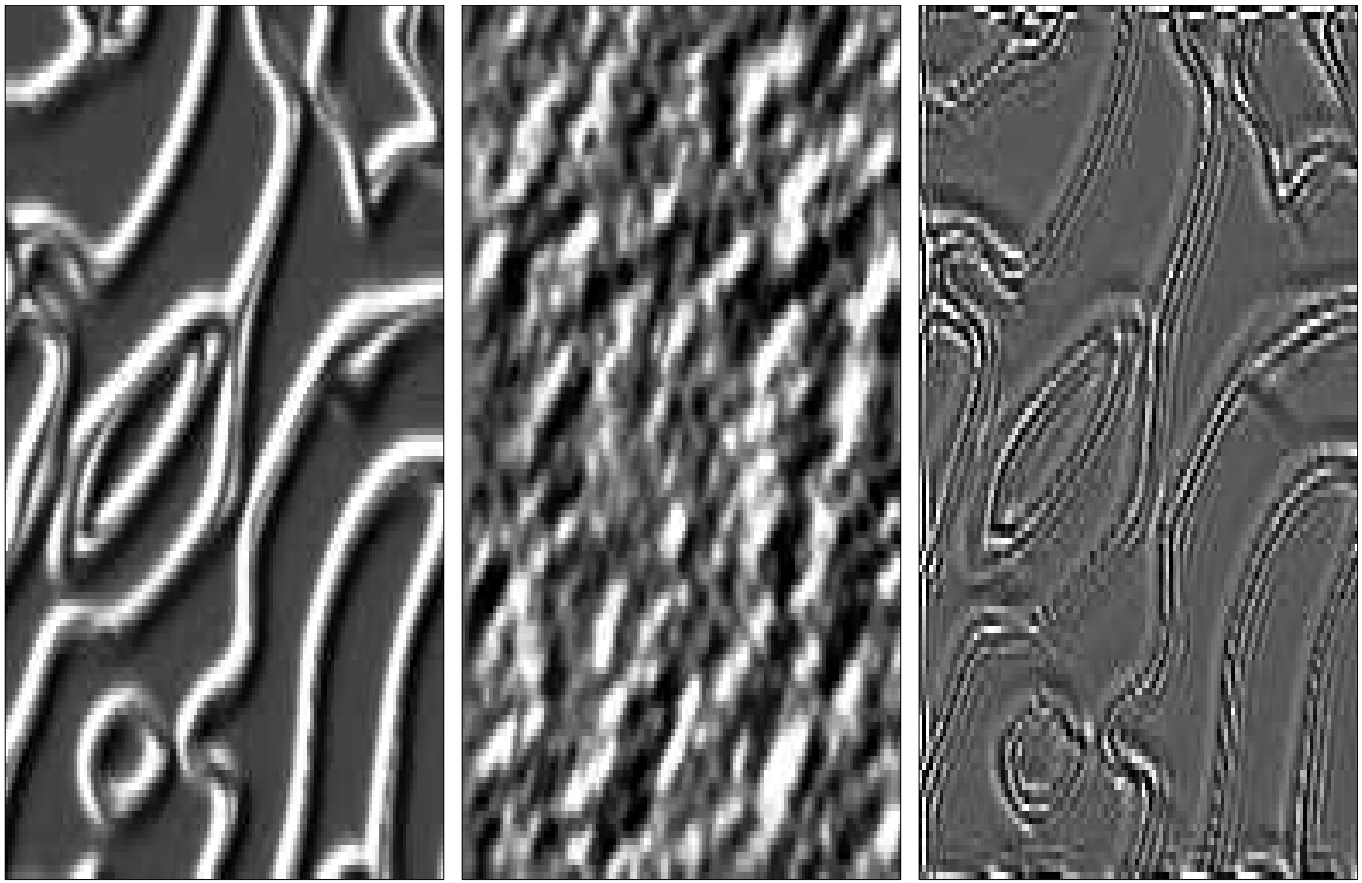


Figure 6.8: Ridges. A spectacular failure of the stationarity assumption. All dips are present but in different locations. Never-the-less, the ridges have been sharpened by the deconvolution. mda-ridges [ER]

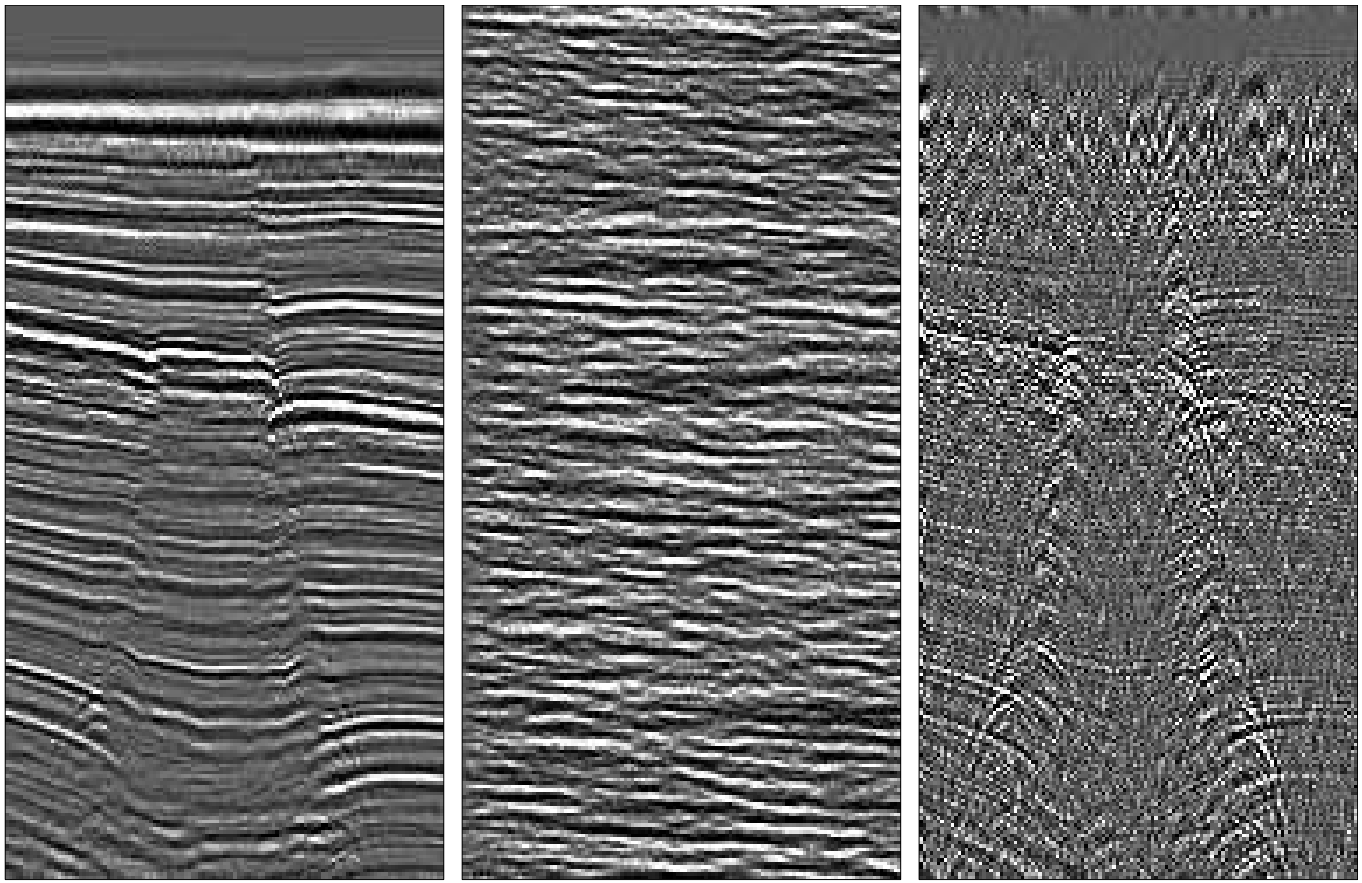


Figure 6.9: Gulf of Mexico seismic section, modeled, and deconvolved. Do you see any drilling prospects in the simulated data? In the deconvolution, the strong horizontal layering is suppressed giving a better view of the hyperbolas. The decon filter is the same 10×10 used on the everyday textures. mda-WGstack [ER]

6.3.5. Seismic field data examples

Figures 6.10-6.13 are based on exploration seismic data from the Gulf of Mexico deep water. A ship carries an air gun and tows a streamer with some hundreds of geophones. First we look at a single pop of the gun. We use all the geophone signals to create a single 1-D PEF for the time axis. This changes the average temporal frequency spectrum as shown in Figure 6.10. Signals from 60 Hz to 120 Hz are boosted substantially. The raw data has evidently been prepared with strong filtering against signals below about 8 Hz. The PEF attempts to recover these signals, mostly unsuccessfully, but it does boost some energy near the 8 Hz cutoff. Choosing a longer filter would flatten the spectrum further. The big question is, "Has the PEF improved the appearance of the data?"

The data itself from the single pop, both before and after PE-filtering is shown in Figure 6.11. For reasons of esthetics of human perception I have chosen to display a mirror image of the PEF'ed data. To see a blink movie of superposition of before-and-after images you need the electronic book. We notice that signals of high temporal frequencies indeed have the expected hyperbolic behavior in space. Thus, these high-frequency signals are wavefields, not mere random noise.

Given that all visual (or audio) displays have a bounded range of amplitudes,

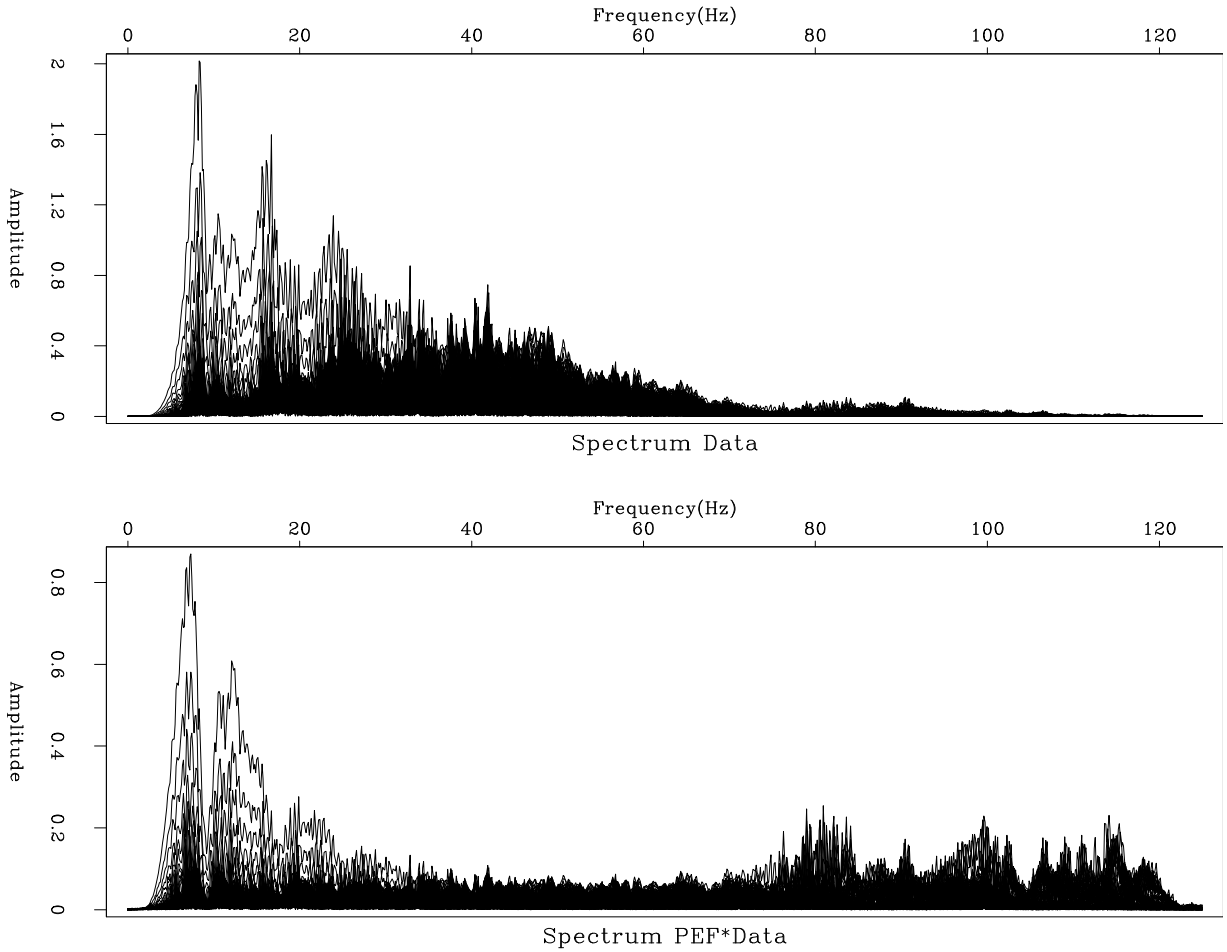


Figure 6.10: ω spectrum of a shot gather of Figure 6.11 before and after 1-D decon with a 30 point filter. mda-antoinedecon1 [ER]

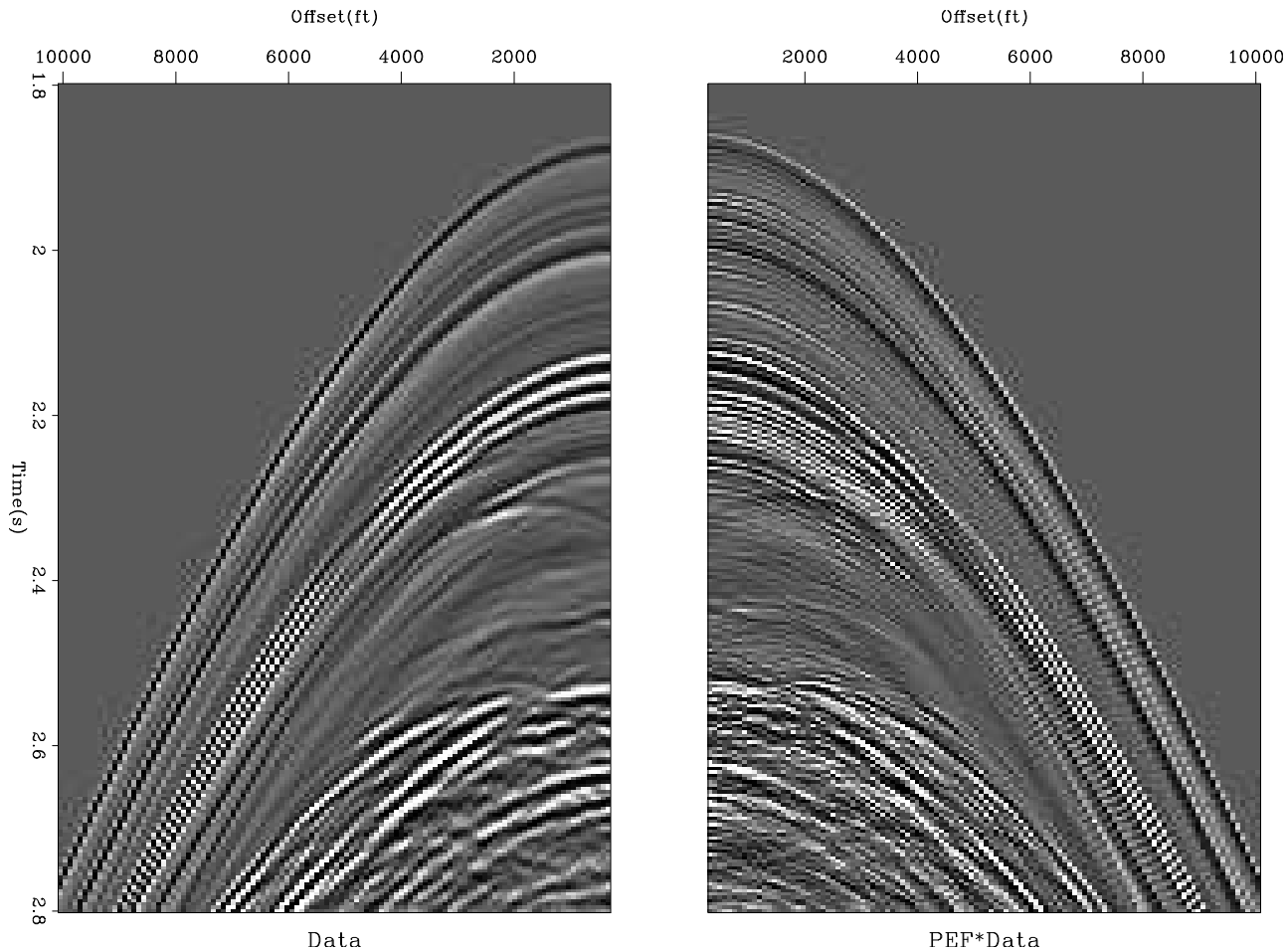


Figure 6.11: Raw data with its mirror. Mirror had 1-D PEF applied, 30 point filter. mda-antoinedecon [ER,M]

increasing the frequency content (bandwidth) means that we will need to turn down the amplification so we do not wish to increase the bandwidth unless we are adding signal.

> Increasing the spectral bandwidth always requires us to diminish the gain.

The same ideas but with a two-dimensional PEF are in Figure 6.12 (the same data but with more of it squeezed onto the page.) As usual, the raw data is dominated by events arriving later at greater distances. After the PEF, we tend to see equal energy in dips in all directions. We have strongly enhanced the "backscattered" energy, those events that arrive later at *shorter* distances.

Figure 6.13 shows echos from the all shots, the nearest receiver on each shot. This picture of the earth is called a "near-trace section." This earth picture shows us why there is so much backscattered energy in Figure 6.12 (which is located at the left side of Figure 6.13). The backscatter comes from any of the many of near-vertical faults.

We have been thinking of the PEF as a tool for shaping the spectrum of a display. But does it have a physical meaning? What might it be? Referring back to the beginning of the chapter we are inclined to regard the PEF as the convolution of

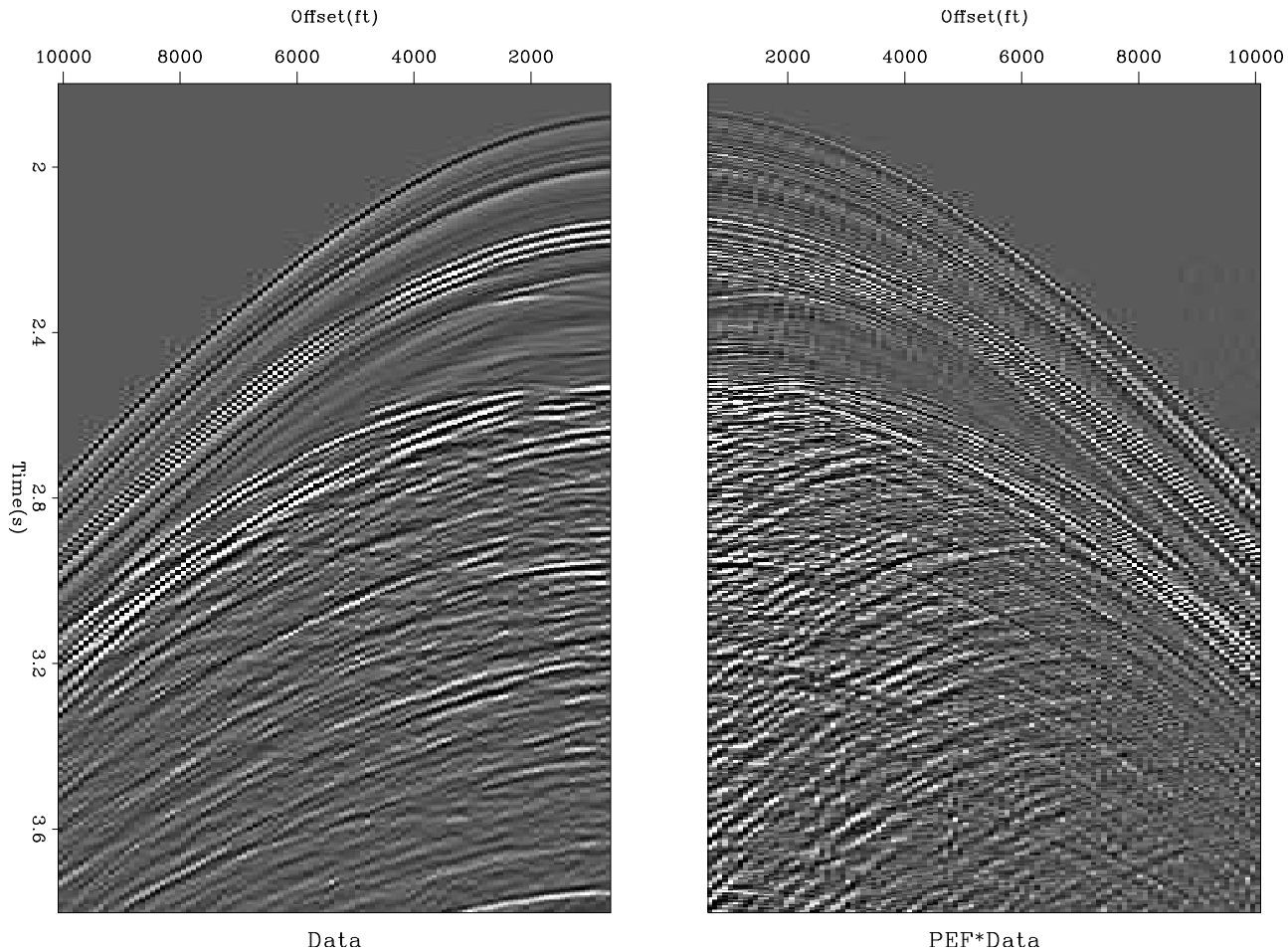


Figure 6.12: A 2-D filter (here 20×5) brings out the backscattered energy. mda-antoinedecon2 [ER,M]

the source waveform with some kind of water-bottom response. In Figure 6.12 we used many different shot-receiver separations. Since each different separation has a different response (due to differing moveouts) the water bottom reverberation might average out to be roughly an impulse. Figure 6.12 is a different story. Here for each shot location, the distance to the receiver is constant. Designing a single channel PEF we can expect the PEF to contain both the shot waveform and the water bottom layers because both are nearly identical in all the shots. We would rather have a PEF that represents only the shot waveform (and perhaps a radiation pattern).

Let us consider how we might work to push the water-bottom reverberation out of the PEF. This data is recorded in water 600 meters deep. A consequence is that the sea bottom is made of fine-grained sediments that settled very slowly and rather similarly from place to place. In shallow water the situation is different. The sands near estuaries are always shifting. Sedimentary layers thicken and thin. They are said to "on-lap and off-lap." Here I do notice where the water bottom is sloped the layers do thin a little. To push the water bottom layers out of the PEF our idea is to base its calculation not on the raw data, but on the spatial prediction error of the raw data. On a perfectly layered earth a perfect spatial prediction error filter would zero all traces but the first one. Since a 2-D PEF includes spatial prediction as well

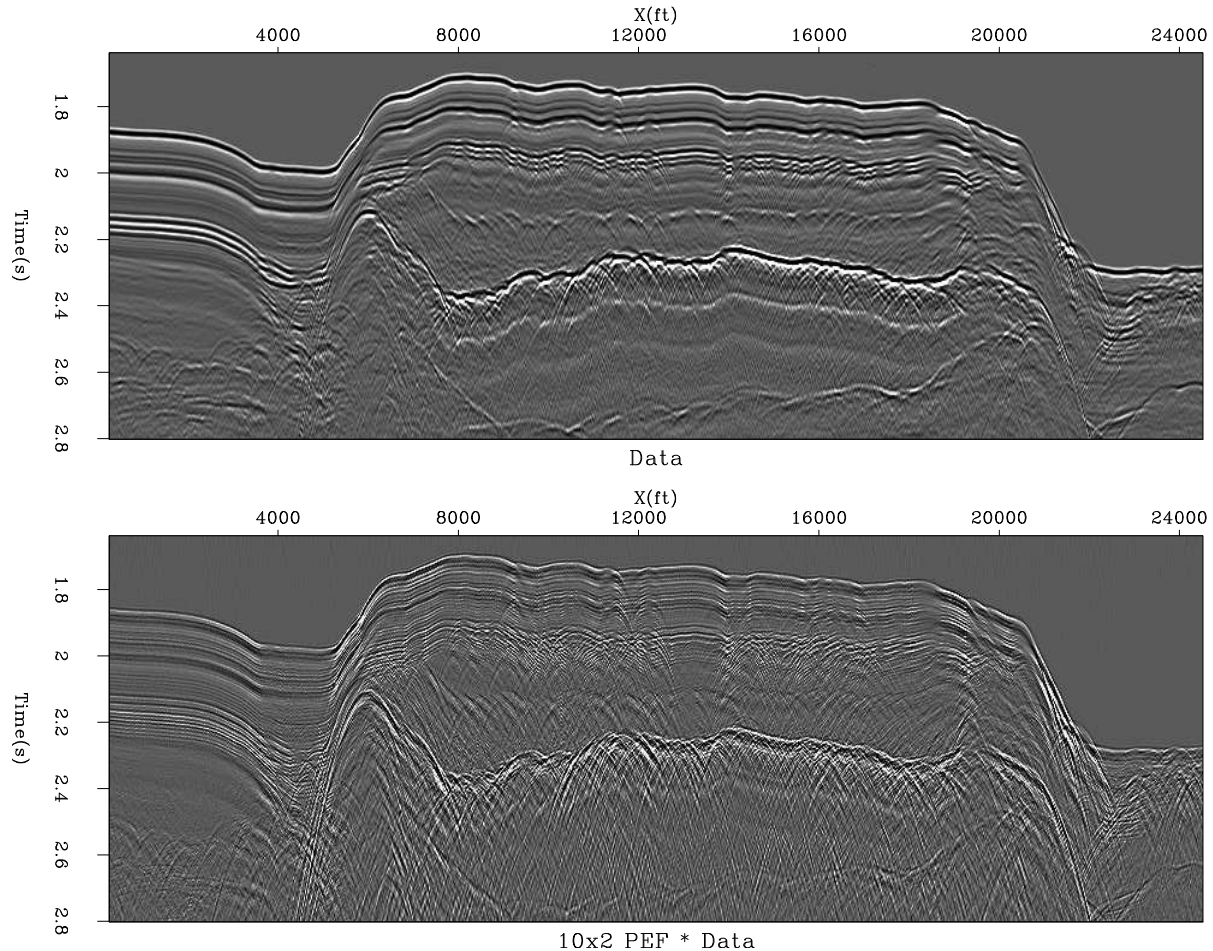


Figure 6.13: Raw data, near-trace section (top). Filtered with a two-channel PEF (bottom). The movie has other shaped filters. mda-antoinedecon3 [ER,M]

as temporal prediction, we can expect it to contain much less of the sea-floor layers than the 1-D PEF. If you have access to the electronic book, you can blink the figure back and forth with various filter shapes.

6.4. PEF ESTIMATION WITH MISSING DATA

If we are not careful, our calculation of the PEF could have the pitfall that it would try to use the missing data to find the PEF, and hence it would get the wrong PEF. To avoid this pitfall, imagine a PEF finder that uses weighted least squares where the weighting function vanishes on those fitting equations that involve missing data. The weighting would be unity elsewhere. Instead of weighting bad results by zero, we simply will not compute them. The residual there will be initialized to zero and never changed. Likewise for the adjoint, these components of the residual will never contribute to a gradient. So now we need a convolution program that produces no outputs where missing inputs would spoil it.

Recall there are two ways of writing convolution, equation (1.4) when we are interested in finding the filter *inputs*, and equation (1.5) when we are interested in finding the *filter itself*. We have already coded equation (1.4), operator `helicon`

```
module hconest {              # masked helix convolution, adjoint is the filter.
use helix
  real, dimension (:), pointer :: x
  type( filter)                :: aa
#% _init( x, aa)
#% _lop( a, y)
    integer  ia, ix, iy
    do ia = 1, size( a) {
         do iy = 1  + aa%lag( ia), size( y) {    if( aa%mis( iy)) cycle
            ix = iy - aa%lag( ia)
             if( adj)    a( ia) +=  y( iy) * x( ix)
             else        y( iy) +=  a( ia) * x( ix)
         }
    }
}
```

Back

/prog:helicon. That operator was useful in missing data problems. Now we want to find a prediction-error filter so we need the other case, equation (1.5), and we need to ignore the outputs that will be broken because of missing inputs. The operator module `hconest` does the job. hconest

We are seeking a prediction error filter $(1, a_1, a_2)$ but some of the data is missing. The data is denoted $\mathbf{y}$ or y_i above and x_i below. Because some of the x_i are missing, some of the regression equations in (6.27) are worthless. When we figure out which are broken, we will put zero weights on those equations.

$$\mathbf{0} \approx \mathbf{r} = \mathbf{WXa} = \begin{bmatrix} w_1 & . & . & . & . & . & . & . \\ . & w_2 & . & . & . & . & . & . \\ . & . & w_3 & . & . & . & . & . \\ . & . & . & w_4 & . & . & . & . \\ . & . & . & . & w_5 & . & . & . \\ . & . & . & . & . & w_6 & . & . \\ . & . & . & . & . & . & w_7 & . \\ . & . & . & . & . & . & . & w_8 \end{bmatrix} \begin{bmatrix} x_1 & 0 & 0 \\ x_2 & x_1 & 0 \\ x_3 & x_2 & x_1 \\ x_4 & x_3 & x_2 \\ x_5 & x_4 & x_3 \\ x_6 & x_5 & x_4 \\ 0 & x_6 & x_5 \\ 0 & 0 & x_6 \end{bmatrix} \begin{bmatrix} 1 \\ a \\ a \end{bmatrix} \tag{6.27}$$

Suppose that x_2 and x_3 were missing or known bad. That would spoil the 2nd, 3rd, 4th, and 5th fitting equations in (6.27). In principle, we want w_2, w_3, w_4 and w_5 to be zero. In practice, we simply want those components of $\mathbf{r}$ to be zero.

What algorithm will enable us to identify the regression equations that have become defective, now that x_2 and x_3 are missing? Take filter coefficients $(a_0, a_1, a_2, \dots)$ to be all ones. Let $\mathbf{d}_{\text{free}}$ be a vector like $\mathbf{x}$ but containing 1's for the missing (or "freely adjustable") data values and 0's for the known data values. Recall our very first definition of filtering showed we can put the filter in a vector and the data in a

matrix or vice versa. Thus **Xa** above gives the same result as **Ax** below.

$$\begin{bmatrix} m_1 \\ m_2 \\ m_3 \\ m_4 \\ m_5 \\ m_6 \\ m_7 \\ m_8 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 2 \\ 2 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \mathbf{A}\mathbf{d}_{\text{free}} \tag{6.28}$$

The numeric value of each m_i tells us how many of its inputs are missing. Where none are missing, we want unit weights $w_i = 1$. Where any are missing, we want zero weights $w_i = 0$. The desired residual under partially missing inputs is computed by module `misinput` /prog:misinput. misinput

```
module misinput {                          # find a mask of missing filter inputs
  use helicon
contains
  subroutine find_mask( known, aa) {
    logical,  intent( in)           :: known(:)
    type( filter)                   :: aa
    real, dimension( size (known)) :: rr, dfre
    integer                         :: stat
    where( known) dfre = 0.
    elsewhere     dfre = 1.
    call helicon_init( aa)
    aa%flt = 1.
    stat = helicon_lop( .false., .false., dfre, rr)
    aa%flt = 0.
    where ( rr > 0.)   aa%mis = .true.
    }
}
```

Back

6.4.1. Internal boundaries to multidimensional convolution

Sometimes we deal with small patches of data. In order that boundary phenomena not dominate the calculation intended in the central region, we need to take care that input data is not assumed to be zero beyond the interval that the data is given.

The two little triangular patches of zeros in the convolution matrix in equation (6.27) describe end conditions where it is assumed that the data y_t vanishes before $t = 1$ and after $t = 6$. Alternately we might not wish to make that assumption. Thus the triangles filled with zeros could be regarded as missing data. In this one-dimensional example, it is easy to see that the filter, say `yy%mis()` should be set to `.TRUE.` at the ends so no output would ever be computed there. We would like to find a general multidimensional algorithm to correctly specify `yy%mis()` around the multidimensional boundaries. This proceeds like the missing data algorithm, i.e. we apply a filter of all ones to a data space template that is taken all zeros except ones at the locations of missing data, in this case y_0, y_{-1} and y_7, y_8. This amounts to surrounding the original data set with some missing data. We need padding the size of the filter on all sides. The padded region would be filled with ones (designating missing inputs). Where the convolution output is nonzero, there `yy%mis()` is set to `.TRUE.` denoting an output with missing inputs.

The two-dimensional case is a little more cluttered than the 1-D case but the principle is about the same. Figure 6.14 shows a larger input domain, a 5×3 filter, and a smaller output domain. There are two things to notice. First, sliding the filter everywhere inside the outer box, we get outputs (under the 1 location) only in the inner box. Second, (the adjoint idea) crosscorrelating the inner and outer boxes gives us the 3×5 patch of information we use to build the filter coefficients. We need to be careful not to assume that signals vanish outside the region where they are defined. In a later chapter we will break data spaces into overlapping patches, separately analyze the patches, and put everything back together. We do this because crosscorrelations change with time and they are handled as constant in short time windows. There we must be particularly careful that zero signal values not be presumed outside of the small volumes; otherwise the many edges and faces of the many small volumes can overwhelm the interior that we want to study.

In practice, the input and output are allocated equal memory, but the output residual is initialized to zero everywhere and then not computed except where shown in figure 6.14. Below is module bound to build a selector for filter outputs that should never be examined or even computed (because they need input data from outside the given data space). Inputs are a filter aa and the size of its cube na

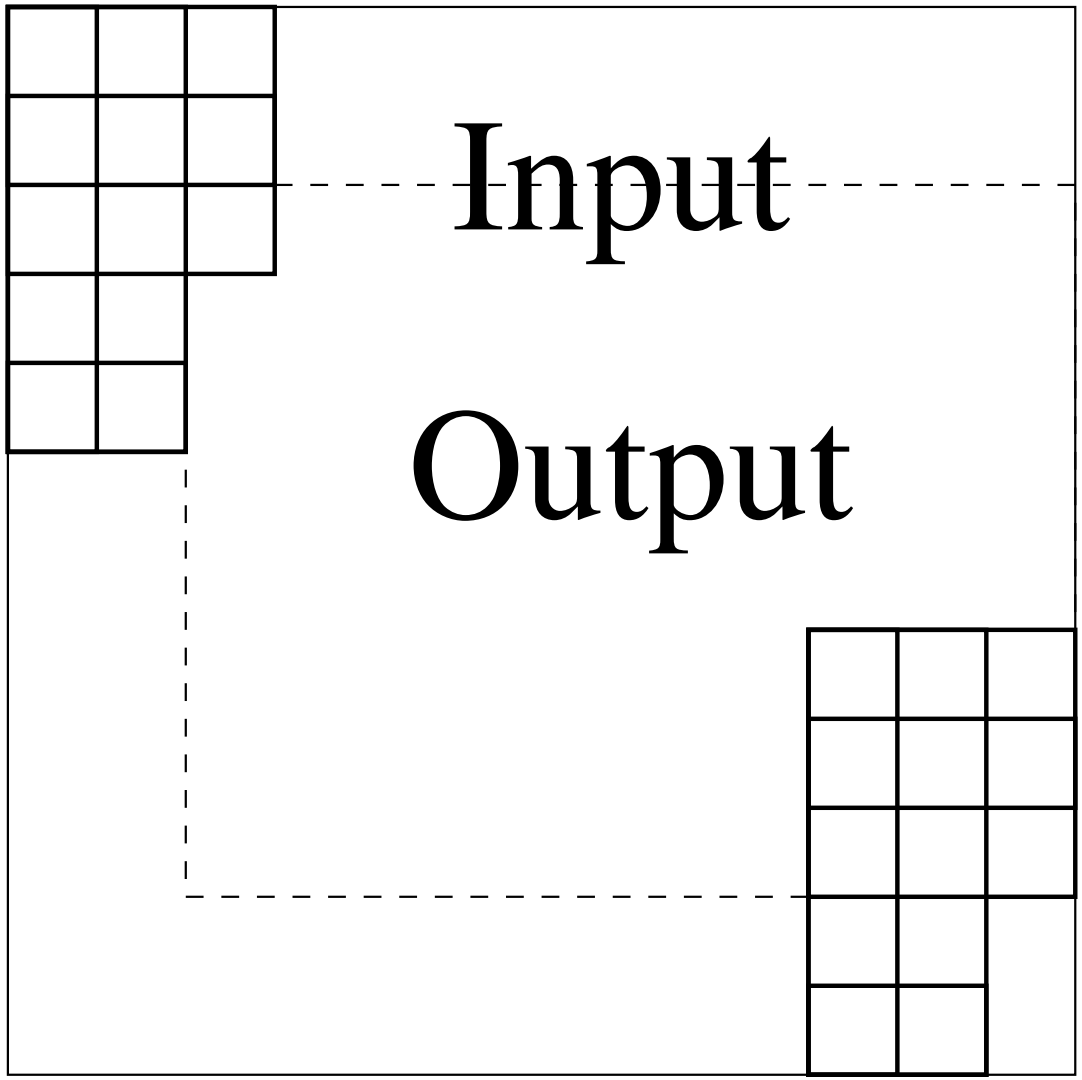


Figure 6.14: Domain of inputs and outputs of a two-dimensional filter like a PEF. mda-rabdomain [NR]

```
module bound {          # mark helix filter outputs where input is off data.
use cartesian
use helicon
use regrid
contains
  subroutine boundn ( nold, nd, na, aa) {
    integer, dimension( :), intent( in) :: nold, nd, na     # (ndim)
    type( filter)                       :: aa
    integer, dimension( size( nd))      :: nb, ii
    real,    dimension( :), allocatable :: xx, yy
    integer                             :: iy, my, ib, mb, stat
    nb = nd + 2*na;   mb = product( nb)      # nb is a bigger space to pad into.
    allocate( xx( mb), yy( mb))              # two large spaces, equal size
    xx = 0.                                  #              zeros
    do ib = 1, mb {                          # surround the zeros with many ones
       call line2cart( nb, ib, ii)           # ii( ib)
       if( any( ii <= na  .or.  ii > nb-na))   xx( ib) = 1.
       }
    call helicon_init( aa)                   # give aa pointer to helicon.lop
    call regridn( nold, nb, aa);  aa%flt = 1.           # put all 1's in filter
    stat = helicon_lop( .false., .false., xx, yy)       # apply filter
    call regridn(   nb, nd, aa);  aa%flt = 0.    # remake filter for orig data.
    my = product( nd)
    allocate( aa%mis( my))               # attach missing designation to y_filter
    do iy = 1, my {                      # map from unpadded to padded space
       call line2cart( nd, iy, ii )
       call cart2line( nb,     ii+na, ib )          # ib( iy)
       aa%mis( iy) =             ( yy( ib) > 0.)    # true where inputs missing
       }
    deallocate( xx, yy)
    }
}
```

Back

`= (na(1),na(2),...)`. Also input are two cube dimensions, that of the data last used by the filter `nold` and that of the filter's next intended use `nd`. (`nold` and `nd` are often the same). Module `bound` begins by defining a bigger data space with room for a filter surrounding the original data space `nd` on all sides. It does this by the line `nb=nd+2*na`. Then we allocate two data spaces `xx` and `yy` of the bigger size `nb` and pack many ones in a frame of width `na` around the outside of `xx`. The filter `aa` is also filled with ones. The filter `aa` must be regridded for the bigger `nb` data space (regridding merely changes the lag values of the ones). Now we filter the input `xx` with `aa` getting `yy`. Wherever the output is nonzero, we have an output that has been affected by the boundary. Such an output should not be computed. Thus we allocate the logical mask `aa%mis` (a part of the helix filter definition in module `helix` /prog:helix) and wherever we see a nonzero value of `yy` in the output, we designate the output as depending on missing inputs by setting `aa%mis` to `.true.`. bound

In reality one would set up the boundary conditions with module `bound` before identifying locations of missing data with module `misinput`. Both modules are based on the same concept, but the boundaries are more cluttered and confusing which is why we examined them later.

6.4.2. Finding the prediction-error filter

The first stage of the least-squares estimation is computing the prediction-error filter. The second stage will be using it to find the missing data. The input data space contains a mixture of known data values and missing unknown ones. For the first stage of finding the filter, we generally have many more fitting equations than we need so we can proceed by ignoring the fitting equations that involve missing data values. We ignore them everywhere that the missing inputs hit the filter.

The codes here do not address the difficulty that maybe too much data is missing so that all weights are zero. To add stabilization we could supplement the data volume with a "training dataset" or by a "prior filter". With things as they are, if there is not enough data to specify a prediction-error filter, you should encounter the error exit from `cgstep()` /prog:cgstep . pef

6.5. TWO-STAGE LINEAR LEAST SQUARES

In Chapter 3 and Chapter 5 we filled empty bins by minimizing the energy output from the filtered mesh. In each case there was arbitrariness in the choice of the filter.

```
module pef {              # Find prediction-error filter (helix magic)
  use hconest
  use cgstep_mod
  use solver_smp_mod
contains
  subroutine find_pef( dd, aa, niter) {
    integer,          intent( in)  :: niter        # number of iterations
    type( filter)                  :: aa           # filter
    real,    dimension(:), pointer :: dd           # input data
    call hconest_init( dd, aa)
    call solver_smp(m=aa%flt, d=-dd, Fop=hconest_lop, step-
per=cgstep, niter=niter, m0=aa%flt)
    call cgstep_close()
    }
}
```

Back

Here we find and use the optimum filter, the PEF.

The first stage is that of the previous section, finding the optimal PEF while carefully avoiding using any regression equations that involve boundaries or missing data. For the second stage, we take the PEF as known and find values for the empty bins so that the power out of the prediction-error filter is minimized. To do this we find missing data with module `mis2()` /prog:mis2 .

This two-stage method avoids the nonlinear problem we would otherwise face if we included the fitting equations containing both free data values and free filter values. Presumably, after two stages of linear least squares we are close enough to the final solution that we could switch over to the full nonlinear setup described near the end of this chapter.

The synthetic data in Figure 6.15 is a superposition of two plane waves of different directions, each with a random (but low-passed) waveform. After punching a hole in the data, we find that the lost data is pleasingly restored, though a bit weak near the side boundary. This imperfection could result from the side-boundary behavior of the operator or from an insufficient number of missing-data iterations.

The residual selector in Figure 6.15 shows where the filter output has valid inputs. From it you can deduce the size and shape of the filter, namely that it matches

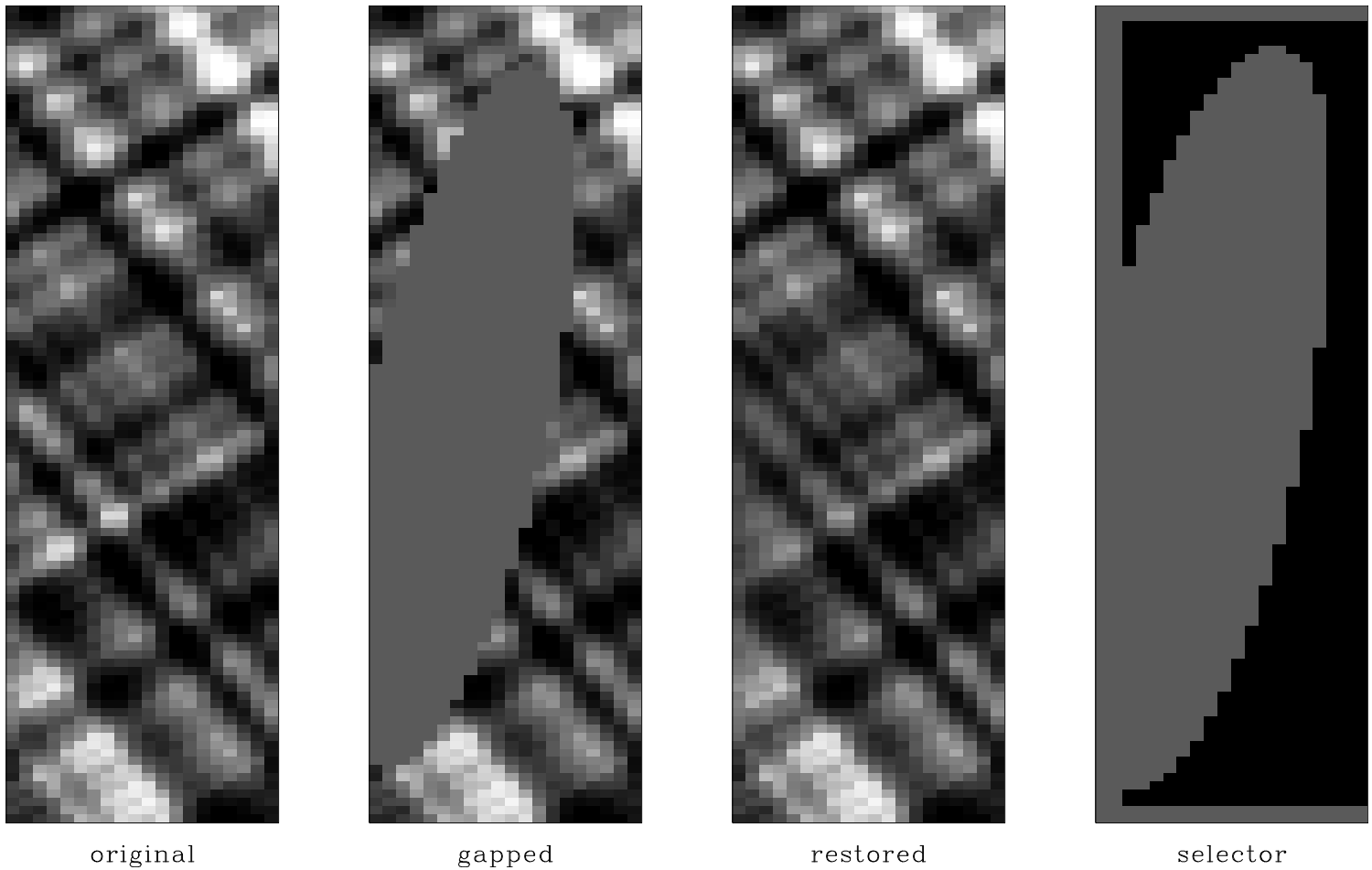


Figure 6.15: Original data (left), with a zeroed hole, restored, residual selector (right). mda-hole90 [ER,M]

up with Figure 6.14. The ellipsoidal hole in the residual selector is larger than that in the data because we lose regression equations not only at the hole, but where any part of the filter overlaps the hole.

The results in Figure 6.15 are essentially perfect representing the fact that that synthetic example fits the conceptual model perfectly. Before we look at the many examples in Figures 6.16-6.19 we will examine another gap-filling strategy.

6.5.1. Adding noise (Geostat)

In chapter 3 we restored missing data by adopting the philosopy of minimizing the energy in filtered output. In this chapter we learned about an optimum filter for this task, the prediction-error filter (PEF). Let us name this method the "minimum noise" method of finding missing data.

A practical problem with the minimum-noise method is evident in a large empty hole such as in Figures 6.16- 6.17. In such a void the interpolated data diminishes greatly. Thus we have not totally succeeded in the goal of "hiding our data acquisition footprint" which we would like to do if we are trying to make pictures of the earth and not pictures of our data acquisition footprint.

What we will do next is useful in some applications but not in others. Misunderstood or misused it is rightly controversial. We are going to fill the empty holes with something that looks like the original data but really isn't. I will distinguish the words "synthetic data" (that derived from a physical model) from "simulated data" (that manufactured from a statistical model). We will fill the empty holes with simulated data like what you see in the center panels of Figures 6.3-6.9. We will add just enough of that "wall paper noise" to keep the variance constant as we move into the void.

Given some data **d**, we use it in a filter operator **D**, and as described with equation (6.27) we build a weighting function **W** that throws out the broken regression equations (ones that involve missing inputs). Then we find a PEF **a** by using this regression.

$$\mathbf{0} \quad \approx \quad \mathbf{r} \quad = \quad \mathbf{WDa} \tag{6.29}$$

Because of the way we defined **W**, the "broken" components of **r** vanish. We need to know the variance σ of the nonzero terms. It can be expressed mathematically in a couple different ways. Let **1** be a vector filled with ones and let $\mathbf{r}^2$ be a vector

containing the squares of the components of $\mathbf{r}$.

$$\sigma \quad = \quad \sqrt{\frac{1}{N}\sum_{i}^{N} r_i^2} \quad = \quad \sqrt{\frac{\mathbf{1}'\mathbf{W}\mathbf{r}^2}{\mathbf{1}'\mathbf{W}\mathbf{1}}} \tag{6.30}$$

Let us go to a random number generator and get a noise vector $\mathbf{n}$ filled with random numbers of variance σ. We'll call this the "added random noise". Now we solve this new regression for the data space $\mathbf{d}$ (both known and missing)

$$\mathbf{0} \quad \approx \quad \mathbf{r} \quad = \quad \mathbf{A}\mathbf{d} - \mathbf{n} \tag{6.31}$$

keeping in mind that known data is constrained (as detailed in chapter 3).

To understand why this works, consider first the training image, a region of known data. Although we might think that the data defines the white noise residual by $\mathbf{r} = \mathbf{A}\mathbf{d}$, we can also imagine that the white noise determines the data by $\mathbf{d} = \mathbf{A}^{-1}\mathbf{r}$. Then consider a region of wholly missing data. This data is determined by $\mathbf{d} = \mathbf{A}^{-1}\mathbf{n}$. Since we want the data variance to be the same in known and unknown locations, naturally we require the variance of $\mathbf{n}$ to match that of $\mathbf{r}$.

A very minor issue remains. Regression equations may have all of their required input data, some of it, or none of it. Should the $\mathbf{n}$ vector add noise to every

regression equation? First, if a regression equation has all its input data that means there are no free variables so it doesn't matter if we add noise to that regression equation because the constraints will overcome that noise. I don't know if I should worry about how *many* inputs are missing for each regression equation.

It is fun making all this interesting "wall paper" noticing where it is successful and where it isn't. We cannot help but notice that it seems to work better with the genuine geophysical data than it does with many of the highly structured patterns. Geophysical data is expensive to acquire. Regrettably, we have uncovered a technology that makes counterfeiting much easier. Examples are in Figures 6.16-6.19. In the electronic book, the right-side panel of each figure is a movie, each panel being derived from different random numbers.

The seismic data in Figure 6.19 illustrates a fundamental principle: In the restored hole we do not see the same spectrum as we do on the other panels. This is because the hole is filled, not with all frequencies (or all slopes) but with those that are most predictable. The filled hole is devoid of the unpredictable noise that is a part of all real data.

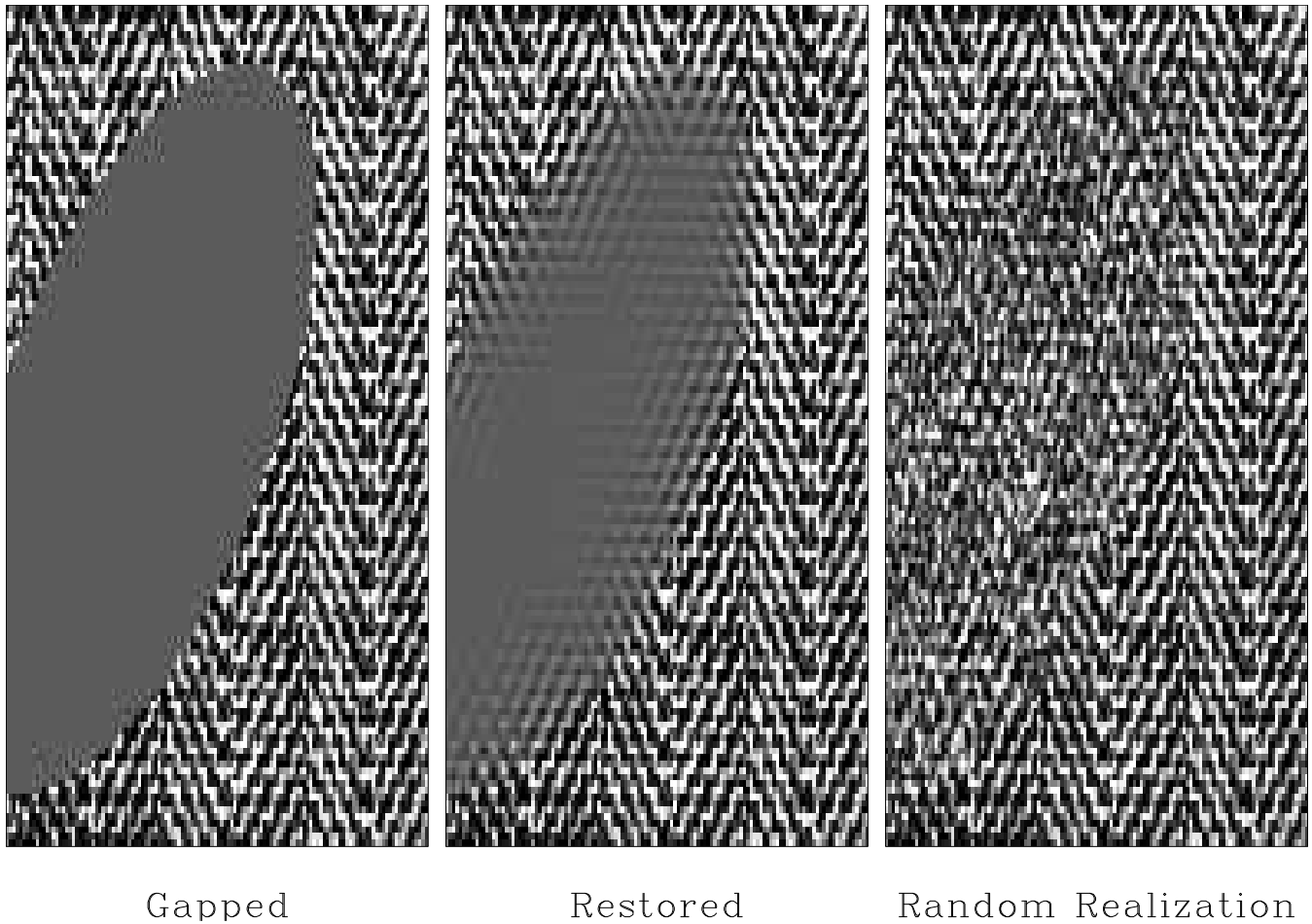


Figure 6.16: The herringbone texture is a patchwork of two textures. We notice that data missing from the hole tends to fill with the texture at the edge of the hole. The spine of the herring fish, however, is not modeled at all. mda-herr-hole-fillr [ER,M]

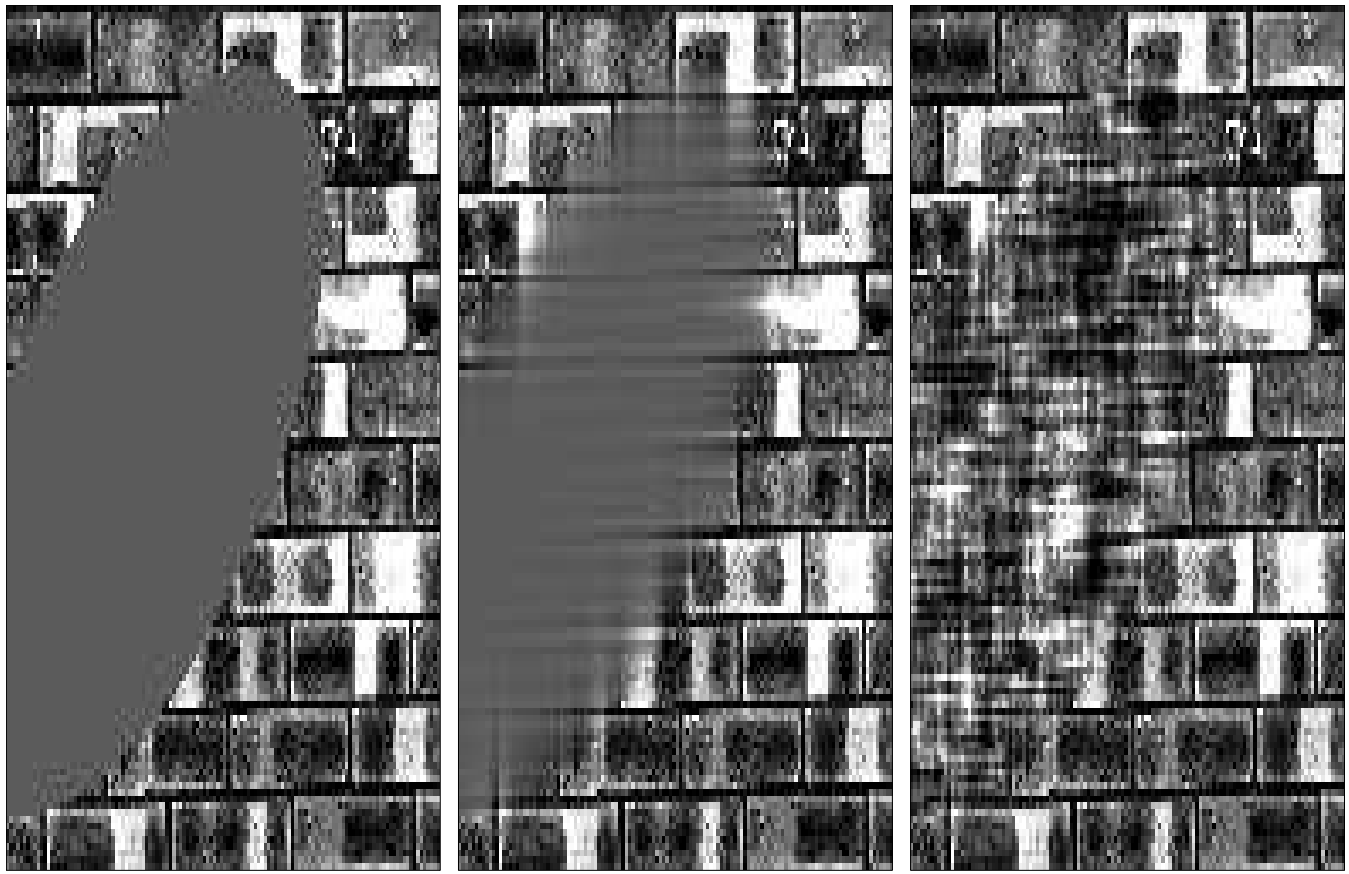


Figure 6.17: The brick texture has a mortar part (both vertical and horizontal joins) and a brick surface part. These three parts enter the empty area but do not end where they should. mda-brick-hole-fillr [ER,M]

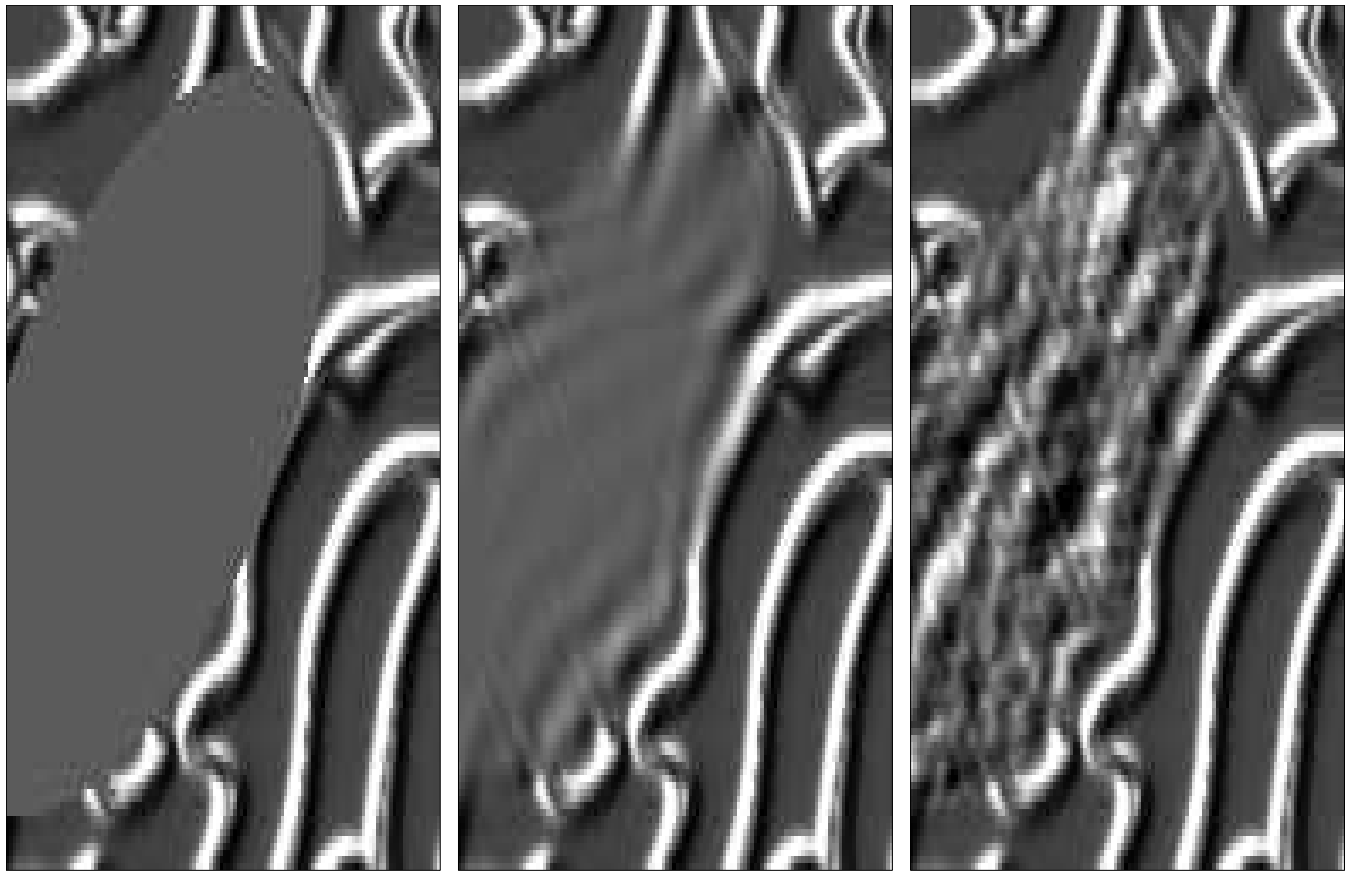


Figure 6.18: The theoretical model is a poor fit to the ridge data since the prediction must try to match ridges of all possible orientations. This data requires a broader theory which incorporates the possibility of nonstationarity (space variable slope). mda-ridges-hole-fillr [ER,M]

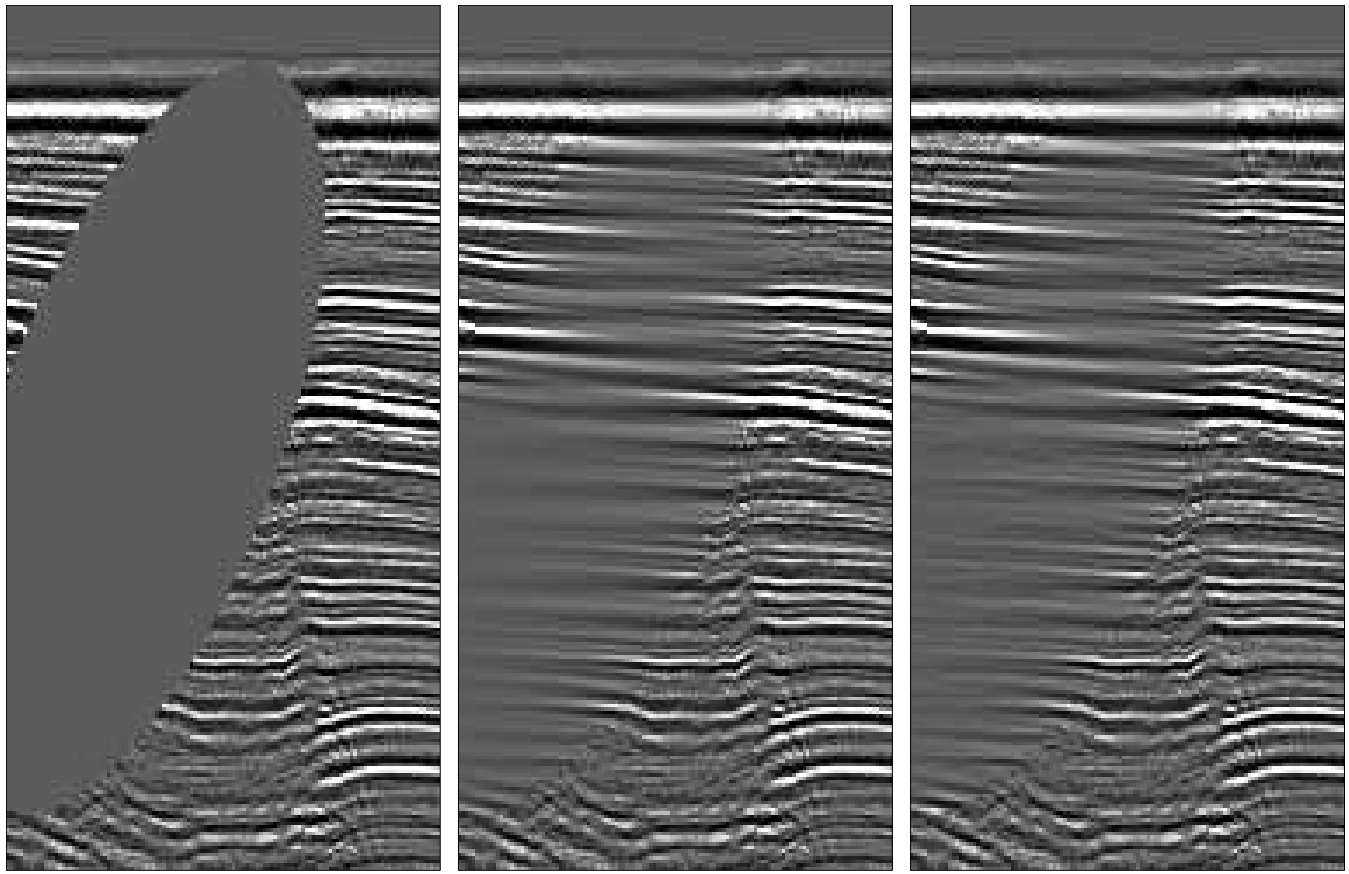


Figure 6.19: Filling the missing seismic data. The imaging process known as "migration" would suffer diffraction artifacts in the gapped data that it would not suffer on the restored data. mda-WGstack-hole-fillr [ER,M]

6.5.2. Inversions with geostat

In geophysical estimation (inversion) we use model styling (regularization) to handle the portion of the model that is not determined by the data. This results in the addition of minimal noise. Alternately, like in Geostatistics, we could make an assumption of statistical stationarity and add much more noise so the signal variance in poorly determined regions matches that in well determined regions. Here is how to do this. Given the usual data fitting and model styling goals

$$\mathbf{0} \approx \mathbf{Lm} - \mathbf{d} \tag{6.32}$$

$$\mathbf{0} \approx \mathbf{Am} \tag{6.33}$$

We introduce a sample of random noise $\mathbf{n}$ and fit instead these regressions

$$\mathbf{0} \approx \mathbf{Lm} - \mathbf{d} \tag{6.34}$$

$$\mathbf{0} \approx \mathbf{Am} - \mathbf{n} \tag{6.35}$$

Of course you get a different solution for each different realization of the random noise. You also need to be a little careful to use noise $\mathbf{n}$ of the appropriate variance. Figure 6.20 shows a result on the SeaBeam data. Bob Clapp developed this idea

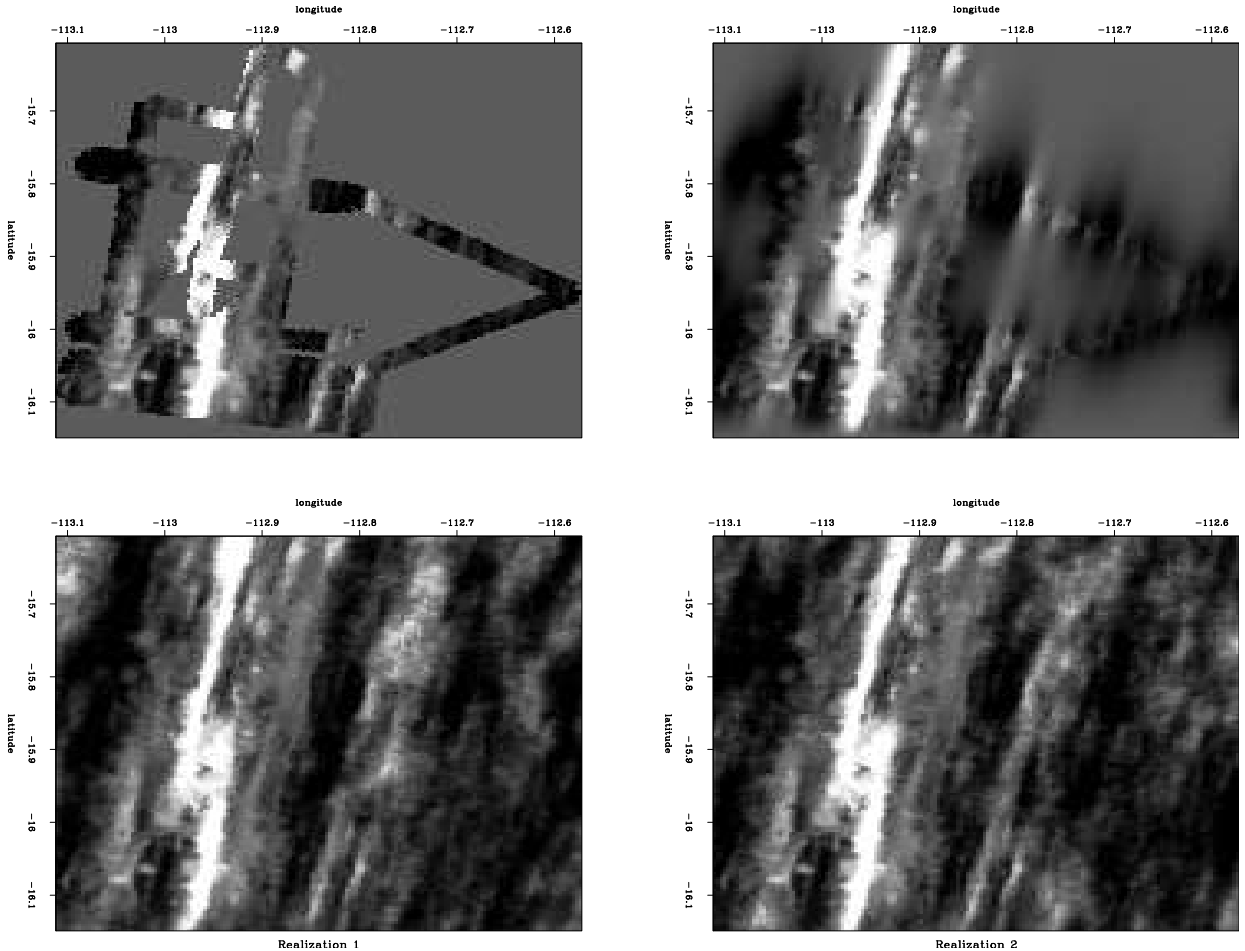


Figure 6.20: Top left is binned data. Top right extends the data with a PEF. The bottom two panels add appropriately colored random noise in the regions of missing data. mda-bobsea [ER,M]

at SEP and also applied it to interval velocity estimation, the example of Figures 5.1-5.3.

6.5.3. Infill of 3-D seismic data from a quarry blast

Finding missing data (filling empty bins) requires use of a filter. Because of the helix, the codes work in spaces of all dimensions.

An open question is how many conjugate-direction iterations are needed in missing-data programs. When estimating filters, I set the iteration count `niter` at the number of free filter parameters. Theoretically, this gives me the exact solution but sometimes I run double the number of iterations to be sure. The missing-data estimation, however is a completely different story. The number of free parameters in the missing-data estimation, could be very large. This often implies impractically long compute times for the exact solution. In practice I experiment carefully with `niter` and hope for the best. I find that where gaps are small, they fill in quickly. Where the gaps are large, they don't, and more iterations are required. Where the gaps are large is where we must experiment with preconditioning.

Figure 6.21 shows an example of replacing missing data by values predicted

from a 3-D PEF. The data was recorded at Stanford University with a 13×13 array of independent recorders. The figure shows 12 of the 13 lines each of length 13. Our main goal was to measure the ambient night-time noise. By morning about half the recorders had dead batteries but the other half recorded a wave from a quarry blast. The raw data was distracting to look at because of the many missing traces so I interpolated it with a small 3-D filter. That filter was a PEF.

6.5.4. Imposing prior knowledge of symmetry

Reversing a signal in time does not change its autocorrelation. In the analysis of stationary time series, it is well known (FGDP) that the filter for predicting forward in time should be the same as that for "predicting" backward in time (except for time reversal). When the data samples are short, however, a different filter may be found for predicting forward than for backward. Rather than average the two filters directly, the better procedure is to find the filter that minimizes the sum of power in two residuals. One is a filtering of the original signal, and the other is a filtering of a time-reversed signal, as in equation (6.36), where the top half of the equations represent prediction-error predicting forward in time and the second half

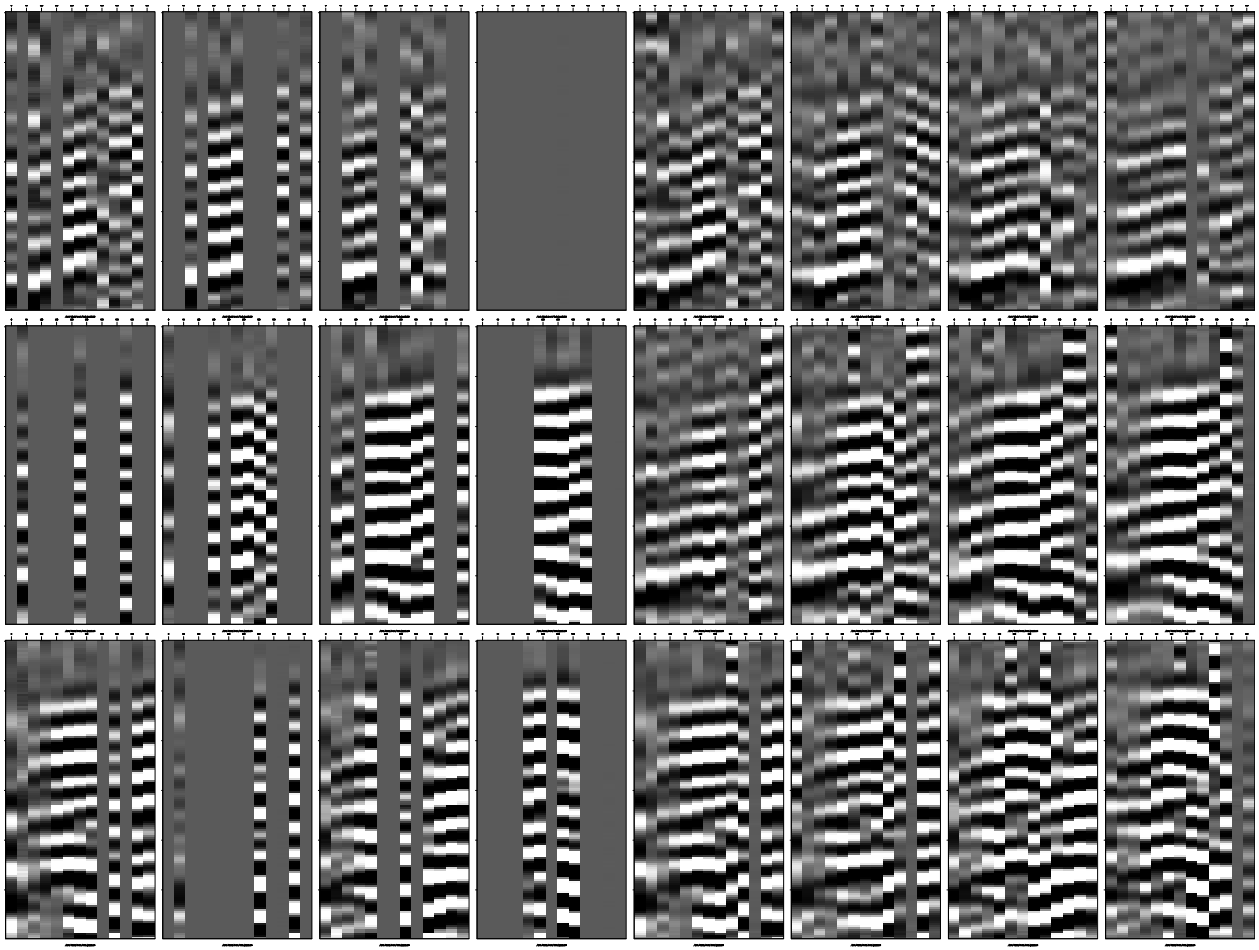


Figure 6.21: The left 12 panels are the inputs. The right 12 panels are outputs. mda-passfill90 [ER,M]

is prediction backward.

$$\left[\begin{array}{c} r_1 \\ r_2 \\ r_3 \\ r_4 \\ \hline r_5 \\ r_6 \\ r_7 \\ r_8 \end{array}\right] = \left[\begin{array}{ccc} y_3 & y_2 & y_1 \\ y_4 & y_3 & y_2 \\ y_5 & y_4 & y_3 \\ y_6 & y_5 & y_4 \\ \hline y_1 & y_2 & y_3 \\ y_2 & y_3 & y_4 \\ y_3 & y_4 & y_5 \\ y_4 & y_5 & y_6 \end{array}\right] \left[\begin{array}{c} 1 \\ a_1 \\ a_2 \end{array}\right] \tag{6.36}$$

To get the bottom rows from the top rows, we simply reverse the order of all the components within each row. That reverses the input time function. (Reversing the order within a column would reverse the output time function.) Instead of the matrix being diagonals tipping 45° to the right, they tip to the left. We could make this matrix from our old familiar convolution matrix and a time-reversal matrix

$$\left[\begin{array}{cccc} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{array}\right]$$

It is interesting to notice how time-reversal symmetry applies to Figure 6.15. First of all, with time going both forward and backward the residual space gets twice as big. The time-reversal part gives a selector for Figure 6.15 with a gap along the right edge instead of the left edge. Thus, we have acquired a few new regression equations.

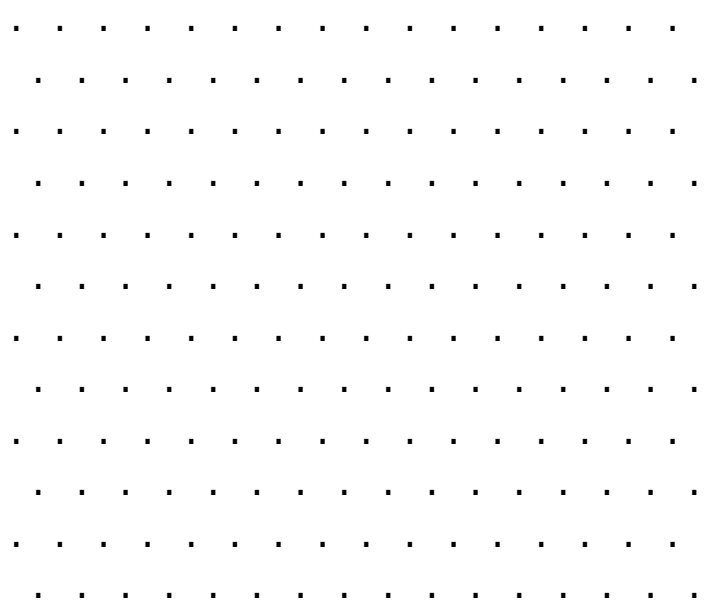
Some of my research codes include these symmetries, but I excluded them here. Nowhere did I see that the reversal symmetry made noticable difference in results, but in coding, it makes a noticeable clutter by expanding the residual to a two-component *residual array*.

Where a data sample grows exponentially towards the boundary, I expect that extrapolated data would diverge too. You can force it to go to zero (or any specified value) at some distance from the body of the known data. To do so, surround the body of data by missing data and surround that by specification of “enough” zeros. “Enough” is defined by the filter length.

6.5.5. Hexagonal coordinates

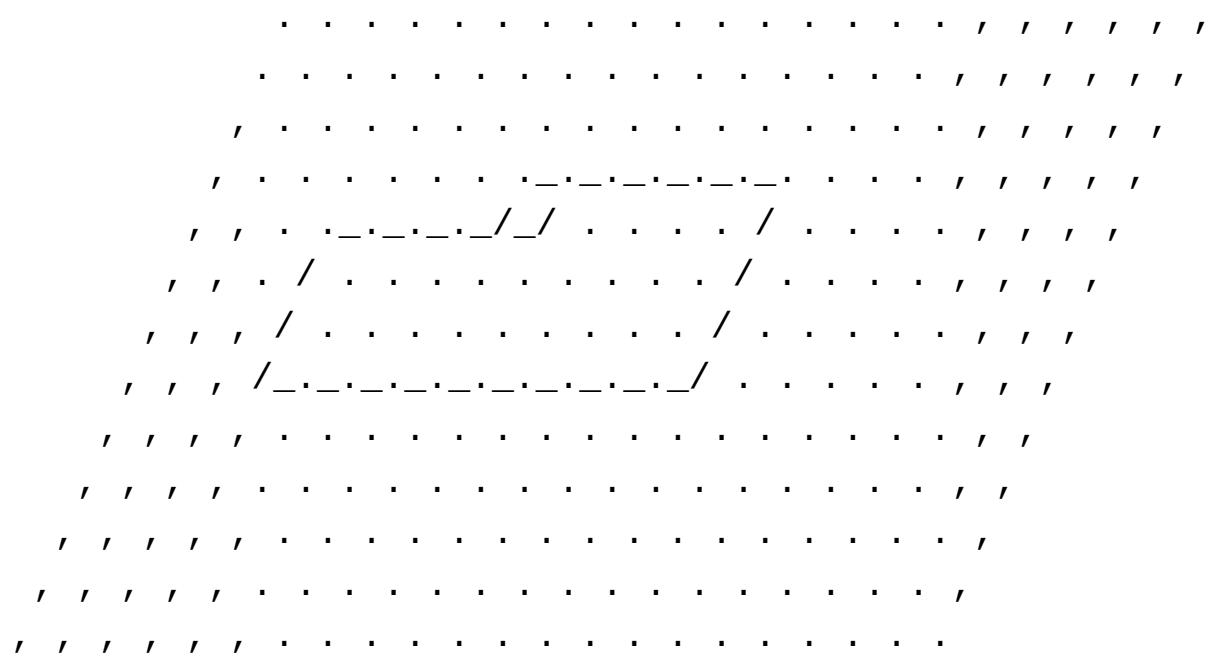
In a two-dimensional plane it seems that the one-sidedness of the PEF could point in any direction. Since we usually have a rectangular mesh, however, we can only do the calculations along the axes so we have only two possibilities, the helix can wrap around the 1-axis, or it can wrap around the 2-axis.

Suppose you acquire data on a hexagonal mesh as below



and some of the data values are missing. How can we apply the methods of this chapter? The solution is to append the given data by more missing data shown by

the commas below.



Now we have a familiar two-dimensional coordinate system in which we can find missing values, as well as perform signal and noise separations as described in a later chapter.

6.6. BOTH MISSING DATA AND UNKNOWN FILTER

Recall the missing-data figures beginning with Figure 3.3. There the filters were taken as known, and the only unknowns were the missing data. Now, instead of having a predetermined filter, we will solve for the filter along with the missing data. The principle we will use is that the output power is minimized while the filter is constrained to have one nonzero coefficient (else all the coefficients would go to zero). We will look first at some results and then see how they were found.

In Figure 6.22 the filter is constrained to be of the form $(1, a_1, a_2)$. The result is pleasing in that the interpolated traces have the same general character as the given values. The filter came out slightly different from the $(1, 0, -1)$ that I guessed and tried in Figure 3.7. Curiously, constraining the filter to be of the form $(a_{-2}, a_{-1}, 1)$ in Figure 6.23 yields the same interpolated missing data as in Figure 6.22. I understand that the sum squared of the coefficients of $A(Z)P(Z)$ is the same as that of $A(1/Z)P(Z)$, but I do not see why that would imply the same interpolated data; never the less, it seems to.

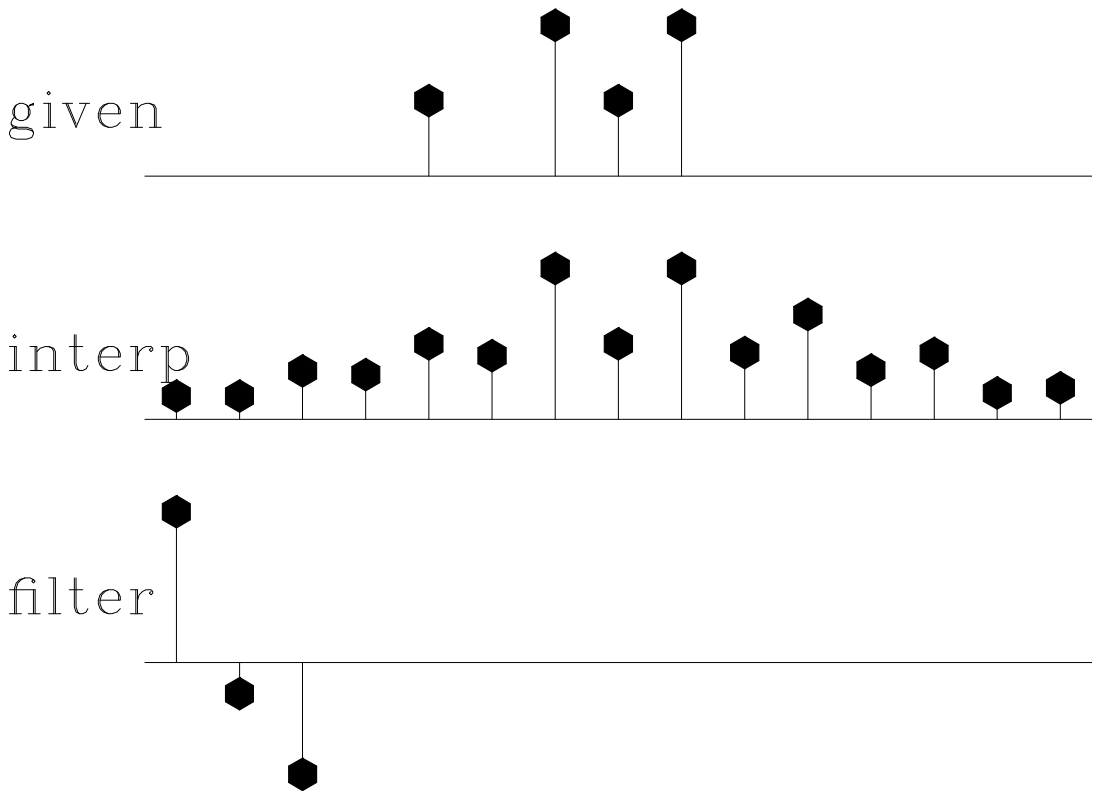


Figure 6.22: Top is known data. Middle includes the interpolated values. Bottom is the filter with the leftmost point constrained to be unity and other points chosen to minimize output power. mda-misif90 [ER]

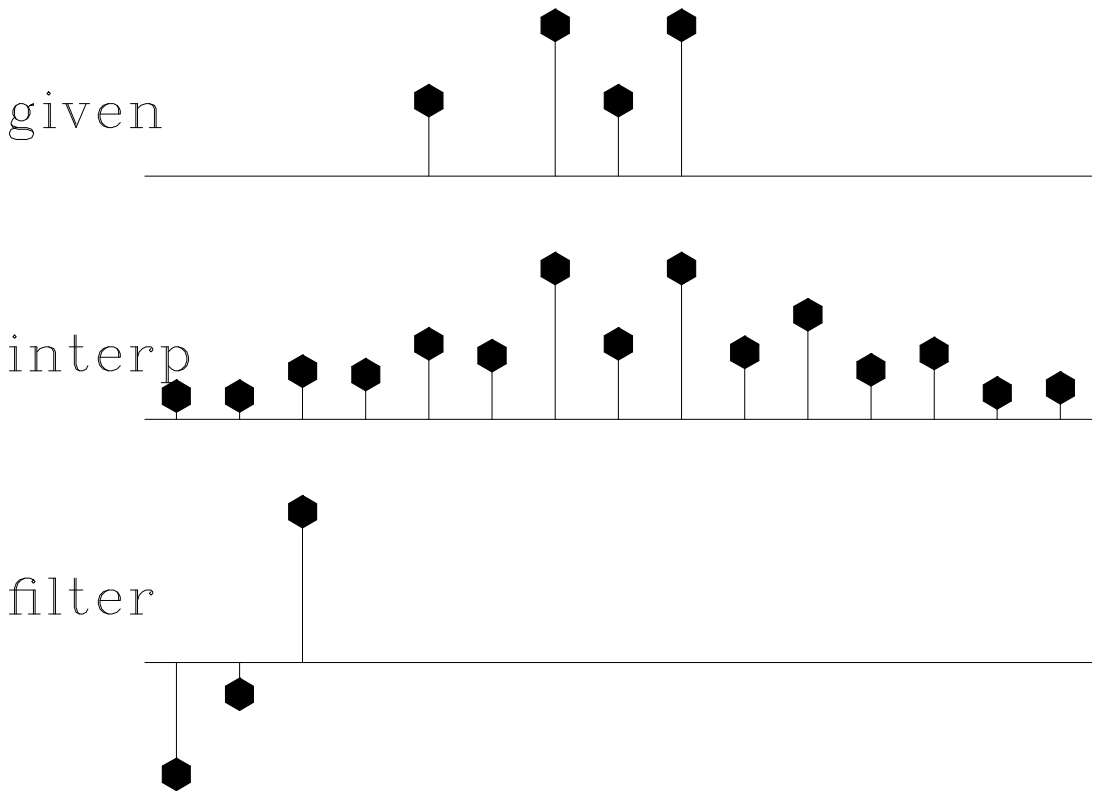


Figure 6.23: The filter here had its rightmost point constrained to be unity—i.e., this filtering amounts to backward prediction. The interpolated data seems to be identical to that of forward prediction. mda-backwards90 [ER]

6.6.1. Objections to interpolation error

In any data interpolation or extrapolation, we want the extended data to behave like the original data. And, in regions where there is no observed data, the extrapolated data should drop away in a fashion consistent with its spectrum determined from the known region.

My basic idea is that the spectrum of the missing data should match that of the known data. This is is the idea that the spectrum should be unchanging from a known region to an unknown region. A technical word to express the idea of spectra not changing is "stationary." This happens with the PEF (one-sided filter) because its spectrum tends to the inverse of that of the known data while that of the unknown data tends to the inverse of that of the PEF. Thus the spectrum of the missing data is the "inverse of the inverse" of the spectrum of the known. The PEF enables us to fill in the missing area with the spectral shape of the known area. (In regions far away or unpredictable, the spectral shape may be the same, but the energy drops to zero.)

On the other hand, the interpolation-error filter, a filter like $(a_{-2}, a_{-1}, 1, a_1, a_2)$, should fail to do the job because it has the wrong spectrum. (I am stating this fact without proof).

To confirm and show these concepts, I prepared synthetic data consisting of a fragment of a damped exponential, and off to one side of it an impulse function. Most of the energy is in the damped exponential. Figure 6.24 shows that the spectrum and the extended data are about what we would expect. From the extrapolated data, it is impossible to see where the given data ends. For comparison, I prepared Figure 6.25. It is the same as Figure 6.24, except that the filter is constrained in the middle. Notice that the extended data does *not* have the spectrum of the given data—the wavelength is much shorter. The boundary between real data and extended data is not nearly as well hidden as in Figure 6.24.

6.6.2. Packing both missing data and filter into a vector

Now let us examine the theory and coding behind the above examples. Define a roughening filter $A(\omega)$ and a data signal $Y(\omega)$ at some stage of interpolation. The fitting goal is $0 \approx A(\omega)Y(\omega)$ where the filter $A(\omega)$ has at least one time-domain coefficient constrained to be nonzero and the data contains both known and missing values. Think of perturbations ΔA and ΔY. We neglect the nonlinear term $\Delta A \, \Delta Y$

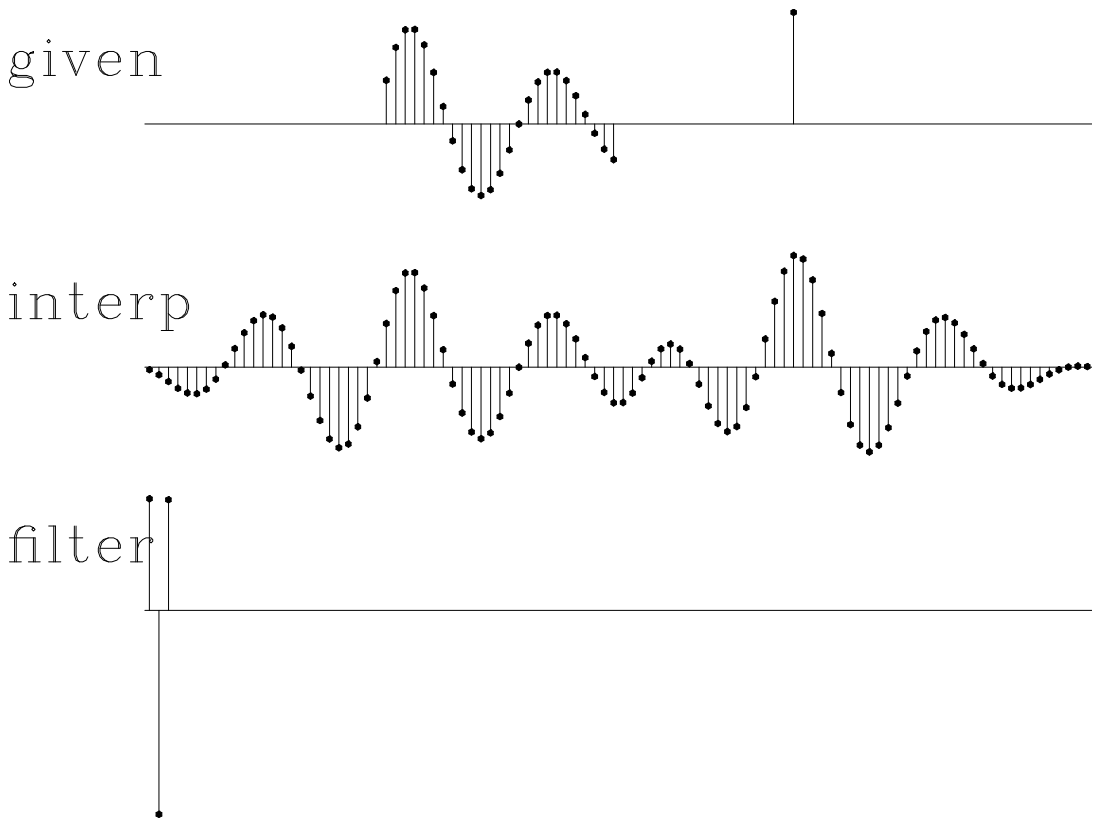


Figure 6.24: Top is synthetic data with missing portions. Middle includes the interpolated values. Bottom is the filter, a *prediction-error* filter which may look symmetric but is not quite. mda-exp90 [ER]

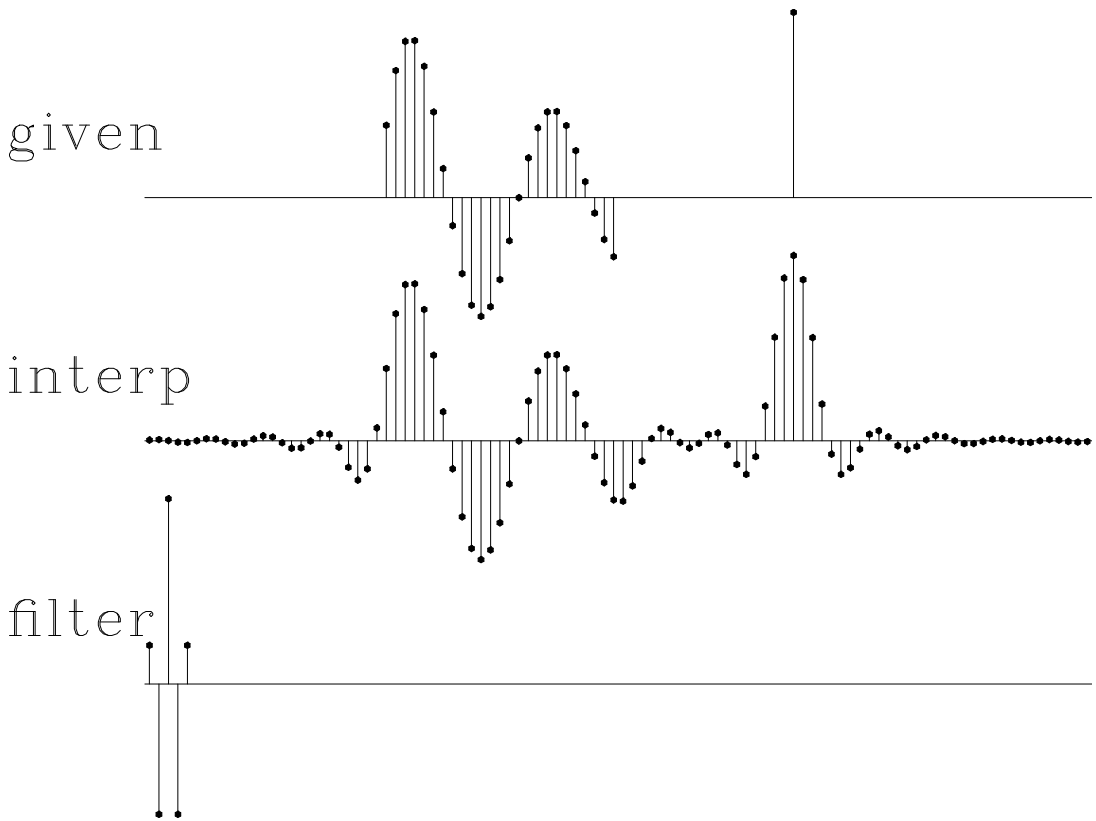


Figure 6.25: Top is the same synthetic data. Middle includes the interpolated values. Bottom is the filter, an *interpolation-error* filter. mda-center90 [ER]

as follows:

$$0 \approx (A + \Delta A)(Y + \Delta Y) \tag{6.37}$$

$$0 \approx A\Delta Y + Y\Delta A + AY + \Delta A\Delta Y \tag{6.38}$$

$$0 \approx A\Delta Y + Y\Delta A + AY \tag{6.39}$$

Let us use matrix algebraic notation to rewrite the fitting goals (6.39). For this we need mask matrices (diagonal matrices with ones on the diagonal where variables are free and zeros where they are constrained i.e., where $\Delta a_i = 0$ and $\Delta y_i = 0$). The free-mask matrix for missing data is denoted **J** and that for the PE filter is **K**. The fitting goal (6.39) becomes

$$\mathbf{0} \approx \mathbf{AJ}\Delta\mathbf{y} + \mathbf{YK}\Delta\mathbf{a} + (\mathbf{Ay} \text{ or } \mathbf{Ya}) \tag{6.40}$$

Defining the original residual as $\bar{\mathbf{r}} = \mathbf{Ay}$ this becomes

$$\mathbf{0} \approx \begin{bmatrix} \mathbf{AJ} & \mathbf{YK} \end{bmatrix} \begin{bmatrix} \Delta\mathbf{y} \\ \Delta\mathbf{a} \end{bmatrix} + \bar{\mathbf{r}} \tag{6.41}$$

For a 3-term filter and a 7-point data signal, the fitting goal (6.40) becomes

$$\begin{bmatrix} a_0 & . & . & . & . & . & . & y_0 & . & . \\ a_1 & a_0 & . & . & . & . & . & y_1 & y_0 & . \\ a_2 & a_1 & a_0 & . & . & . & . & y_2 & y_1 & y_0 \\ . & a_2 & a_1 & a_0 & . & . & . & y_3 & y_2 & y_1 \\ . & . & a_2 & a_1 & a_0 & . & . & y_4 & y_3 & y_2 \\ . & . & . & a_2 & a_1 & a_0 & . & y_5 & y_4 & y_3 \\ . & . & . & . & a_2 & a_1 & a_0 & y_6 & y_5 & y_4 \\ . & . & . & . & . & a_2 & a_1 & . & y_6 & y_5 \\ . & . & . & . & . & . & a_2 & . & . & y_6 \end{bmatrix} \begin{bmatrix} \mathbf{J} & \mathbf{0} \\ \mathbf{0} & \mathbf{K} \end{bmatrix} \begin{bmatrix} \Delta y_0 \\ \Delta y_1 \\ \Delta y_2 \\ \Delta y_3 \\ \Delta y_4 \\ \Delta y_5 \\ \Delta y_6 \\ \hline \Delta a_0 \\ \Delta a_1 \\ \Delta a_2 \end{bmatrix} + \begin{bmatrix} \bar{r}_0 \\ \bar{r}_1 \\ \bar{r}_2 \\ \bar{r}_3 \\ \bar{r}_4 \\ \bar{r}_5 \\ \bar{r}_6 \\ \bar{r}_7 \\ \bar{r}_8 \end{bmatrix} \tag{6.42}$$

Recall that $\bar{r}_t$ is the convolution of a_t with y_t, namely, $\bar{r}_0 = y_0 a_0$ and $\bar{r}_1 = y_0 a_1 + y_1 a_0$, etc. To optimize this fitting goal we first initialize $\mathbf{a} = (1, 0, 0, \cdots)$ and then put zeros in for missing data in $\mathbf{y}$. Then we iterate over equations (6.43) to (6.47).

$$\mathbf{r} \quad \longleftarrow \quad \mathbf{A}\mathbf{y} \tag{6.43}$$

$$\begin{bmatrix} \Delta\mathbf{y} \\ \Delta\mathbf{a} \end{bmatrix} \quad \longleftarrow \quad \begin{bmatrix} \mathbf{J}'\mathbf{A}' \\ \mathbf{K}'\mathbf{Y}' \end{bmatrix} \mathbf{r} \tag{6.44}$$

$$\Delta\mathbf{r} \quad \longleftarrow \quad \begin{bmatrix} \mathbf{AJ} & \mathbf{YK} \end{bmatrix} \begin{bmatrix} \Delta\mathbf{y} \\ \Delta\mathbf{a} \end{bmatrix} \tag{6.45}$$

$$\mathbf{y} \quad \longleftarrow \quad \text{cgstep}(\mathbf{y}, \Delta\mathbf{y}) \tag{6.46}$$

$$\mathbf{a} \quad \longleftarrow \quad \text{cgstep}(\mathbf{a}, \Delta\mathbf{a}) \tag{6.47}$$

This is the same idea as all the linear fitting goals we have been solving, except that now we recompute the residual $\mathbf{r}$ inside the iteration loop so that as convergence is achieved (*if* it is achieved), the neglected nonlinear term $\Delta A \Delta Y$ tends to zero.

My initial research proceeded by linearization like (6.39). Although I ultimately succeeded, I had enough difficulties that I came to realize that linearization is dangerous. When you start "far enough" from the correct solution the term $\Delta A \, \Delta Y$ might not actually be small enough. You don't know how small is small, because these are not scalars but operators. Then the solution may not converge to the minimum you want. Your solution will depend on where you start from. I no

longer exhibit the nonlinear solver `missif` until I find a real data example where it produces noticibly better results than multistage linear-least squares.

The alternative to linearization is two-stage linear least squares. In the first stage you estimate the PEF; in the second you estimate the missing data. If need be, you can re-estimate the PEF using all the data both known and missing (downweighted if you prefer).

If you don't have enough regression equations because your data is irregularly distributed, then you can use binning. Still not enough? Try coarser bins. The point is that nonlinear solvers will not work unless you begin close enough to the solution, and the way to get close is by arranging first to solve a sensible (though approximate) linearized problem. Only as a last resort, after you have gotten as near as you can, should you use the nonlinear least-squares techniques.

6.7. LEVELED INVERSE INTERPOLATION

Eighteenth- and nineteenth- century mathematics literature gives us many methods of interpolating functions. These classical methods are generally based on polynomials. The user specifies some order of polynomial and the theory works out the

coefficients. Today our interest is in both interpolating and extrapolating wavefields (which are solutions to low order differential equations) and we use methods that are much better behaved than polynomials when extrapolating data, methods which behave acceptably when faced with contradictory data values, and methods which also apply in two and three dimensions.

In Chapter 3, subroutine `invint1()` /prog:invint2 solved the problem of inverse linear interpolation, which is, given scattered data points, to find a function on a uniform mesh from which linear interpolation gives the scattered data points. To cope with regions having no data points, the subroutine requires an input roughening filter. This is a bit like specifying a differential equation to be satisfied between the data points. The question is, how should we choose a roughening filter? The importance of the roughening filter grows as the data gets sparser or as the mesh is refined.

Figures 6.22-6.25 suggest that the choice of the roughening filter need not be subjective, nor a priori, but that the prediction-error filter (PEF) is the ideal roughening filter. Spectrally, the PEF tends to the inverse of its input hence its output tends to be “level”. Missing data that is interpolated with this “leveler” tends to have the spectrum of given data.

6.7.1. Test results for leveled inverse interpolation

Figures 6.26 and 6.27 show the same example as in Figures 3.13 and 3.14. What is new here is that the proper PEF is not given but is determined from the data. Figure 6.26 was made with a three-coefficient filter $(1, a_1, a_2)$ and Figure 6.27 was made with a five-coefficient filter $(1, a_1, a_2, a_3, a_4)$. The main difference in the figures is where the data is sparse. The data points in Figures 3.13, 6.26 and 6.27 are samples from a sinusoid.

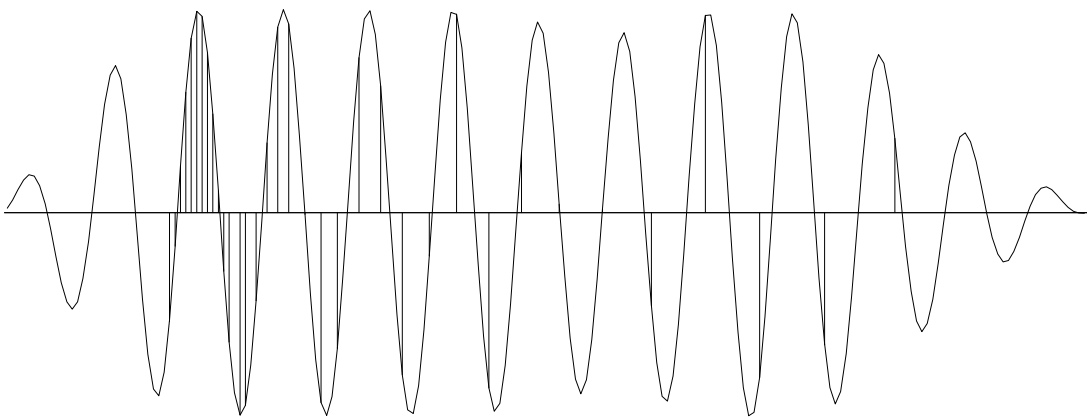


Figure 6.26: Interpolating with a three-term filter. The interpolated signal is fairly monofrequency. mda-subsine390 [ER,M]

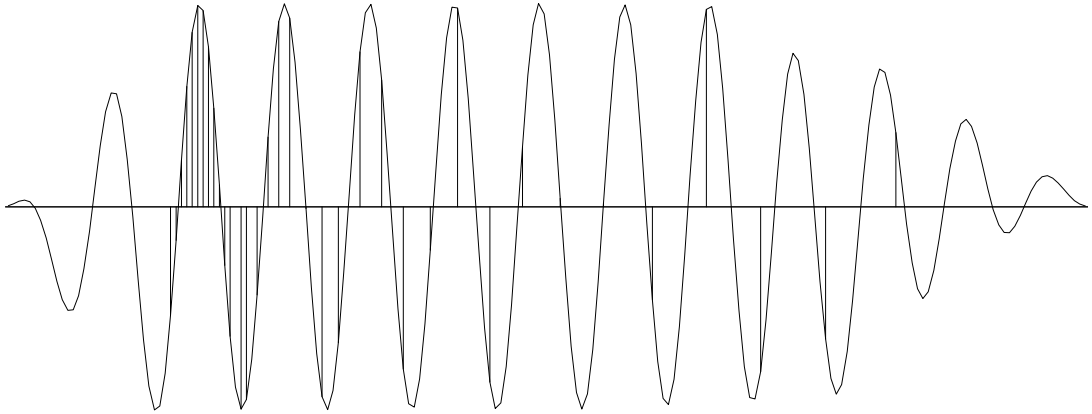


Figure 6.27: Interpolating with a five term filter. mda-subsine590 [ER,M]

Comparing Figures 3.13 and 3.14 to Figures 6.26 and 6.27 we conclude that by finding and imposing the prediction-error filter while finding the model space, we have interpolated beyond aliasing in data space.

6.7.2. Analysis for leveled inverse interpolation

Here we see how the interpolation beyond aliasing was done. The first "statement of wishes" is that the observational data **d** should result from a linear interpolation **L** of the uniformly sampled model space **m**; that is, $\mathbf{0} \approx \mathbf{Lm} - \mathbf{d}$. Expressing this

as a change $\Delta\mathbf{m}$ gives the fitting goal in terms of the model change, $\mathbf{0} \approx \mathbf{L}\Delta\mathbf{m} + (\mathbf{Lm} - \mathbf{d}) = \mathbf{L}\Delta\mathbf{m} + \mathbf{r}$. The second wish is really an assertion that a good way to find missing parts of a function (the model space) is to solve for the function and its PEF at the same time. We are merging the fitting goal (3.15) for irregularly sampled data with the fitting goal (6.42) for finding the prediction-error filter.

$$\mathbf{0} \quad \approx \quad \mathbf{r}_d \quad = \quad \mathbf{L}\Delta\mathbf{m} + (\mathbf{Lm} - \mathbf{d}) \tag{6.48}$$

$$\mathbf{0} \quad \approx \quad \mathbf{r}_m \quad = \quad \mathbf{A}\Delta\mathbf{m} + \mathbf{MK}\Delta\mathbf{a} + (\mathbf{Am} \text{ or } \mathbf{Ma}) \tag{6.49}$$

Writing this out in full for 3 data points and 6 model values on a uniform mesh and a PEF of 3 terms, we have

$$\left[\begin{array}{cccccc|ccc} .8 & .2 & . & . & . & . & & & \\ . & . & 1 & . & . & . & & & \\ . & . & . & . & .5 & .5 & & & \\ \hline a_0 & . & . & . & . & . & m_0 & . & . \\ a_1 & a_0 & . & . & . & . & m_1 & m_0 & . \\ a_2 & a_1 & a_0 & . & . & . & m_2 & m_1 & m_0 \\ . & a_2 & a_1 & a_0 & . & . & m_3 & m_2 & m_1 \\ . & . & a_2 & a_1 & a_0 & . & m_4 & m_3 & m_2 \\ . & . & . & a_2 & a_1 & a_0 & m_5 & m_4 & m_3 \\ . & . & . & . & a_2 & a_1 & . & m_5 & m_4 \\ . & . & . & . & . & a_2 & . & . & m_5 \end{array}\right] \left[\begin{array}{cc} \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{K} \end{array}\right] \left[\begin{array}{c} \Delta m_0 \\ \Delta m_1 \\ \Delta m_2 \\ \Delta m_3 \\ \Delta m_4 \\ \Delta m_5 \\ \Delta m_6 \\ \hline \Delta a_0 \\ \Delta a_1 \\ \Delta a_2 \end{array}\right] + \left[\begin{array}{c} r_{d0} \\ r_{d1} \\ r_{d2} \\ \hline r_{m0} \\ r_{m1} \\ r_{m2} \\ r_{m3} \\ r_{m4} \\ r_{m5} \\ r_{m6} \\ r_{m7} \end{array}\right] \tag{6.50}$$

where r_m is the convolution of the filter a_t and the model m_t, where r_d is the data misfit $\mathbf{r} = \mathbf{Lm} - \mathbf{d}$, and where $\mathbf{K}$ was defined in equation (6.11).

Before you begin to use this nonlinear fitting goal, you need some starting

guesses for $\mathbf{m}$ and $\mathbf{a}$. The guess $\mathbf{m} = 0$ is satisfactory (as explained later). For the first guess of the filter, I suggest you load it up with $\mathbf{a} = (1, -2, 1)$ as I did for the examples here.

6.7.3. Seabeam: theory to practice

I provide here a more fundamental theory for dealing with the Seabeam data. I originally approached the data in this more fundamental way, but with time, I realized that I paid a high price in code complexity, computational speed, and reliability. The basic problem is that the elegant theory requires a good starting model which can only come from the linearized theory. I briefly recount the experience here, because the fundamental theory is interesting and because in other applications, you will face the challenge of sorting out the fundamental features from the essential features.

The linear-interpolation operator carries us from a uniform mesh to irregularly distributed data. Fundamentally we seek to solve the inverse problem to go the other direction. A nonlinear approach to filling in the missing data is suggested by the one-dimensional examples in Figures 6.26–6.27, where the PEF and the missing

data are estimated simultaneously. The nonlinear approach has the advantage that it allows for completely arbitrary data positioning, whereas the two-stage linear approach forces the data to be on a uniform mesh and requires there not be too many empty mesh locations.

For the 2-D nonlinear problem, we follow the same approach we used in one dimension, equations (6.48) and (6.49), except that the filtering and the linear interpolations are two dimensional.

I have had considerable experience with this problem on this data set and I can report that bin filling is easier and works much more quickly and reliably. Eventually I realized that the best way to start the nonlinear iteration (6.48-6.49) is with the final result of bin filling. Then I learned that the extra complexity of the nonlinear iteration (6.48-6.49) offers little apparent improvement to the quality of the SeaBeam result. (This is not to say that we should not try more variations on the idea).

Not only did I find the binning method faster, but I found it to be *much* faster (compare a minute to an hour). The reasons for being faster (most important first) are,

1. Binning reduces the amount of data handled in each iteration by a factor of

the average number of points per bin.

2. The 2-D linear interpolation operator adds many operations per data point.

3. Using two fitting goals seems to require more iterations.

(Parenthetically, I later found that helix preconditioning speeds the Seabeam interpolation from minutes to seconds.)

The most serious criticism of the nonlinear approach is that it does not free us from the linearized approaches. We need them to get a "close enough" starting solution to the nonlinear problem. I learned that the iteration (6.48-6.49), like most nonlinear sequences, behaves unexpectedly and badly when you start too far from the desired solution. For example, I often began from the assumed PEF being a Laplacian and the original map being fit from that. Oddly, from this starting location I sometimes found myself stuck. The iteration (6.48-6.49) would not move towards the map we humans consider a better one.

Having said all those bad things about iteration (6.48-6.49), I must hasten to add that with a different type of data set, you might find the results of (6.48-6.49) to be significantly better.

6.7.4. Risky ways to do nonlinear optimization

I have noticed that some geophysicists have adopted a risky method of nonlinear optimization, which is not advocated in the professional optimization literature. This risky method is to linearize a goal (with a multiparameter model space), then optimize the linearized goal, then relinearize, etc. The safer method is to relinearize after each step of CD.

An instructive example I learned about many years ago was earthquake epicenter location. Model space is latitude, longitude, and origin time. When people added a new variable, the depth, the solutions went wild until they learned to restrict the depth to zero until the other three parameters were stabilized. Apparently the instability stems from the fact that depth and origin time affect distant receivers in a similar way.

6.7.5. The bane of PEF estimation

This is the place where I would like to pat myself on the back for having "solved" the problem of missing data. Actually, an important practical problem remains. I've been trying to coax younger, more energetic people to think about it. The problem

arises when there is too much missing data.

> The bane of PEF estimation is too much missing data

Then *all* the regression equations disappear. The nonlinear methods are particularly bad because if they don't have a good enough starting location, they can and do go crazy. My only suggestion is to begin with a linear PEF estimator. Shrink the PEF and coarsen the mesh in model space until you do have enough equations. Starting from there, hopefully you can refine this crude solution without dropping into a local minimum.

Another important practical problem remains, that of nonstationarity. We'll see the beginnings of the solution to that problem in chapter 9.

6.8. MULTIVARIATE SPECTRUM

A common spectrum is the Fourier spectrum. More fundamentally, a spectrum is a decomposition of a model space or data space into components. The components are in some sense independent; more specifically, the components are orthogonal to

one another. Another well-known spectrum is provided by eigenvectors and eigenvalues. In statistical signal processing we handle a third type of spectrum, the multivariate spectrum.

Working in an optimization problem, we begin from residuals between theory and practice. These residuals can be scaled to make new optimization residuals before we start minimizing their energy. What scaling should we use? The scaling can be a simple weighting function or a filter. A filter is simply a weighting function in Fourier space.

The basic idea of common sense, which also comes to us as results proven by Gauss or from the theory of statistical signal processing, is this: The optimization residuals should be roughly of equal scale. This makes sense because squaring magnifies scale, and anything small will be ignored while anything large will dominate. Scaling optimization residuals to be in a common range makes them all equally influential on the final solution. Not only should optimization residuals be of like scale in physical space, they should be of like scale in Fourier space or eigenvector space, or any other space that we might use to represent the optimization residuals. This implies that the optimization residuals should be uncorrelated. If the optimization residuals were correlated, they would have a spectrum that was not white. Not

white means of differing sizes in Fourier space. Residuals should be the same size as one another in physical space, likewise in Fourier space. Thus the optimization residuals should be orthogonal and of unit scale, much like Fourier components or as eigenvectors are orthonormal.

Let us approach the problem backwards. Suppose we have two random variables that we take to be the ideal optimization residuals x_1 and x_2. In reality the two may be few or trillions. In the language of statistics, the optimization residuals are expected to have zero mean, an idea that is formalized by writing $E(x_1) = 0$ and $E(x_2) = 0$. Likewise these ideal optimization residuals have equal energy, $E(x_1^2) = 1$ and $E(x_2^2) = 1$. Finally, these two optimization residuals are uncorrelated, a condition which is written as $E(x_1 x_2) = 0$. The expectation symbol $E()$ is like a summation over many instances of the random variable.

Now suppose there exists a transformation $\mathbf{B}$ from these ideal optimization residuals to two experimental residuals y_1 and y_2, say $\mathbf{y} = \mathbf{Bx}$ where

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \tag{6.51}$$

The experimental residuals y_1 and y_2 are likely to be neither orthogonal nor equal in

energy. From the column vector $\mathbf{y}$, the experimenter can form a square matrix. Let us also allow the experimenter to write the symbol $E()$ to denote summation over many trials or over many sections of data, ranges over time or space, over soundings or over receiver locations. The experimenter writes

$$\mathbf{R} = E(\mathbf{y}\mathbf{y}') \tag{6.52}$$

$$\mathbf{R} = E(\mathbf{B}\mathbf{x}\mathbf{x}'\mathbf{B}') \tag{6.53}$$

Given a random variable r, the expectation of $2r$ is simply $E(2r) = 2E(r)$. The $E()$ symbol is a summation on random variables, but constants like the coefficients of $\mathbf{B}$ pass right through it. Thus,

$$\mathbf{R} = \mathbf{B}\, E(\mathbf{x}\mathbf{x}')\, \mathbf{B}' \tag{6.54}$$

$$\mathbf{R} = \mathbf{B}\, E\left(\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \begin{bmatrix} x_1 & x_2 \end{bmatrix} \right) \mathbf{B}' \tag{6.55}$$

$$\mathbf{R} = \mathbf{B} \begin{bmatrix} E(x_1 x_1) & E(x_1 x_2) \\ E(x_2 x_1) & E(x_2 x_2) \end{bmatrix} \mathbf{B}' \tag{6.56}$$

$$\mathbf{R} = \mathbf{B}\mathbf{B}' \tag{6.57}$$

Given a matrix $\mathbf{R}$, there is a simple well-known method called the Cholesky factorization method that will factor $\mathbf{R}$ into two parts like $\mathbf{B}$ and $\mathbf{B}'$. The method creates for us either an upper or a lower triangular matrix (our choice) for $\mathbf{B}$. You can easily reinvent the Cholesky method if you multiply the symbols for two triangular matrices like $\mathbf{B}$ and $\mathbf{B}'$ and notice the procedure that works backwards from $\mathbf{R}$ to $\mathbf{B}$. The experimenter seeks not $\mathbf{B}$, however, but its inverse, the matrix that takes us from the experimental residuals to the ideal optimization residuals that are uncorrelated and of equal energies. The Cholesky factorization costs N^3 computations, which is about the same as the cost of the matrix inversion of $\mathbf{R}$ or $\mathbf{B}$. For geophysical maps and other functions on Cartesian spaces, the Prediction Error Filter (PEF) accomplishes the same general goal and has the advantage that we have already learned how to perform the operation using operators instead of matrices.

The multivariate spectrum of experimental residuals $\mathbf{y}$ is the matrix $\mathbf{R} = E(\mathbf{y}\mathbf{y}')$. For optimum model finding, the experimental residuals (squared) should be weighted inversely (matrix inverse) by their multivariate spectrum.

If I were a little stronger at analysis (or rhetoric) I would tell you that the optimizers preconditioned variable $\mathbf{p}$ is the statisticians IID (Independent Identically

Distributed) random variable. For stationary (statistically constant) signals and images, $\mathbf{A}_m$ is the model-space PEF. Echo soundings and interval velocity have statistical properties that change with depth. There $\mathbf{A}_m$ is a diagonal weighting matrix (perhaps before or after a PEF).

6.8.1. What should we optimize?

Least-squares problems often present themselves as fitting goals such as

$$\mathbf{0} \quad \approx \quad \mathbf{F}\mathbf{m} - \mathbf{d} \tag{6.58}$$

$$\mathbf{0} \quad \approx \quad \mathbf{m} \tag{6.59}$$

To balance our possibly contradictory goals we need weighting functions. The quadratic form that we should minimize is

$$\min_{m} \quad (\mathbf{F}\mathbf{m} - \mathbf{d})' \mathbf{A}_n' \mathbf{A}_n (\mathbf{F}\mathbf{m} - \mathbf{d}) + \mathbf{m}' \mathbf{A}_m' \mathbf{A}_m \mathbf{m} \tag{6.60}$$

where $\mathbf{A}_n' \mathbf{A}_n$ is the inverse multivariate spectrum of the noise (data-space residuals) and $\mathbf{A}_m' \mathbf{A}_m$ is the inverse multivariate spectrum of the model. In other words, $\mathbf{A}_n$ is

a leveler on the data fitting error and $\mathbf{A}_m$ is a leveler on the model. There is a curious unresolved issue: What is the most suitable constant scaling ratio of $\mathbf{A}_n$ to $\mathbf{A}_m$?

6.8.2. Confusing terminology for data covariance

Confusion often stems from the mean of the data $E(\mathbf{d})$.

An experimentalist would naturally believe that the expectation of the data is solely a function of the data, that it can be estimated by averaging data.

On the other hand, a theoretician's idea of the expectation of the observational data $E(\mathbf{d})$ is that it is the theoretical data $\mathbf{Fm}$, that the expectation of the data $E(\mathbf{d}) = \mathbf{Fm}$ is a function of the model. The theoretician thinks this way because of the idea of noise $\mathbf{n} = \mathbf{Fm} - \mathbf{d}$ as having zero mean.

Seismological data is highly complex but also highly reproducible. In studies like seismology, the world is deterministic but more complicated than our ability to model. Thus, as a practical matter, the discrepancy between observational data and theoretical data is more realistically attributed to the theoretical data. It is not adequately modeled and computed.

This superficial difference in viewpoint becomes submerged to a more subtle

level by statistical textbooks that usually define weighting functions in terms of variances instead of spectra. This is particularly confusing with the noise spectrum $(\mathbf{A}_n'\mathbf{A}_n)^{-1}$. It is often referred to as the "data covariance" defined as $E[(\mathbf{d}-E(\mathbf{d}))(\mathbf{d}-E(\mathbf{d}))']$. Clearly, the noise spectrum is the same as the data covariance only if we accept the theoretician's definition that $E(\mathbf{d})=\mathbf{Fm}$.

There is no ambiguity and no argument if we drop the word "variance" and use the word "spectrum". Thus, (1) the "inverse noise spectrum" is the appropriate weighting for data-space residuals; and (2) the "inverse model spectrum" is the appropriate model-space weighting. Theoretical expositions generally require these spectra to be given as "prior information." In this book we see how, when the model space is a map, we can solve for the "prior information" along with everything else.

The statistical words "covariance matrix" are suggestive and appealing, but I propose not to use them because of the ambiguity of $E(\mathbf{d})$. For example, we understand that people who say "data covariance" intend the "multivariate noise spectrum" but we cannot understand their meaning of "model covariance". They should intend the "multivariate model spectrum" but that implies that $E(\mathbf{m})=\mathbf{0}$, which seems wrong. Avoiding the word "covariance" avoids the problem.

6.8.3. Hermeneutics

Hermeneutics is the study of the methodological principles of interpretation. Historically, it refers to bible study. Never-the-less, it seems entirely appropriate for Geophysical Estimation. If Albert's book is "Inverse Problem Theory" and mine is "Inverse Problem Practice", and if the difference between theory and practice is smaller in theory than it is in practice, then there are two fundamental questions:

1. In theory, what is the difference between theory and practice? In theory, the difference is data error.

2. In practice, what is the difference between theory and practice? One suggestion is that the discrepancy is entirely due to inadequate modeling. It is well known that geophysical data is highly repeatable. The problem is that the modeling neglects far too much.

Here is a perspective drawn from analysis of the human genome: "The problem is that it is possible to use empirical data to calibrate a model that generates simulated data that is similar to the empirical data. The point of using such a calibrated model is to be able to show how strange certain regions are if they don't fit the simulated distribution, which is based on the empirical distribution." In other words,

"inversion" is just the process of calibrating a model. To learn something new we study the *failures* of such models.

Chapter 7

Noisy data

Noise comes in two distinct flavors. First is erratic bursty noise which is difficult to fit into a statistical model. It bursts out of our simple models. To handle this noise we need “robust” estimation procedures which we consider first.

Next is noise that has a characteristic spectrum, temporal spectrum, spatial spectrum, or dip spectrum. Such noise is called “stationary” noise. A special case of stationary noise is low frequency drift of the mean value of a signal.

In real life, we need to handle both bursty noise and stationary noise at the same time. We’ll try that now.

7.1. MEANS, MEDIANS, PERCENTILES AND MODES

Means, medians, and modes are different averages. Given some data values d_i for $i = 1, 2, ..., N$, the arithmetic mean value m_2 is

$$m_2 = \frac{1}{N} \sum_{i=1}^{N} d_i \tag{7.1}$$

It is useful to notice that this m_2 is the solution of the simple fitting problem $d_i \approx m_2$ or $\mathbf{d} \approx m_2$, in other words, $\min_{m_2} \sum_i (m_2 - d_i)^2$ or

$$0 \quad = \quad \frac{d}{dm_2} \sum_{i=1}^{N} (m_2 - d_i)^2 \tag{7.2}$$

The median of the d_i values is found when the values are sorted from smallest to largest and then the value in the middle is selected. The median is delightfully well behaved even if some of your data values happen to be near infinity. Analytically, the median arises from the optimization

$$\min_{m_1} \sum_{i=1}^{N} |m_1 - d_i| \tag{7.3}$$

To see why, notice that the derivative of the absolute value function is the signum function,

$$\text{sgn}(x) \quad = \quad \lim_{\epsilon \longrightarrow 0} \frac{x}{|x| + \epsilon} \tag{7.4}$$

The gradient vanishes at the minimum.

$$0 \quad = \quad \frac{d}{dm_1} \sum_{i=1}^{N} |m_1 - d_i| \tag{7.5}$$

The derivative is easy and the result is a sum of sgn() functions,

$$0 \quad = \quad \sum_{i=1}^{N} \text{sgn}(m_1 - d_i) \tag{7.6}$$

In other words it is a sum of plus and minus ones. If the sum is to vanish, the number of plus ones must equal the number of minus ones. Thus m_1 is greater than half the data values and less than the other half, which is the definition of a median. The mean is said to minimize the ℓ^2 norm of the residual and the median is said to minimize its ℓ^1 norm.

Before this chapter, our model building was all based on the ℓ^2 norm. The median is clearly a good idea for data containing large bursts of noise, but the median is a single value while geophysical models are made from many unknown elements. The ℓ^1 norm offers us the new opportunity to build multiparameter models where

the data includes huge bursts of noise.

Yet another average is the "mode," which is the most commonly occurring value. For example, in the number sequence $(1,1,2,3,5)$ the mode is 1 because it occurs the most times. Mathematically, the mode minimizes the zero norm of the residual, namely $\ell^0 = |m_0 - d_i|^0$. To see why, notice that when we raise a residual to the zero power, the result is 0 if $d_i = m_0$, and it is 1 if $d_i \neq m_0$. Thus, the ℓ^0 sum of the residuals is the total number of residuals less those for which d_i matches m_0. The minimum of $\ell^0(m)$ is the mode $m = m_0$. The zero power function is nondifferentiable at the place of interest so we do not look at the gradient.

$\ell^2(m)$ and $\ell^1(m)$ are convex functions of m (positive second derivative for all m), and this fact leads to the triangle inequalities $\ell^p(a) + \ell^p(b) \geq \ell^p(a+b)$ for $p \geq 1$ and assures slopes lead to a unique (if $p > 1$) bottom. Because there is no triangle inequality for ℓ^0, it should not be called a "norm" but a "measure."

Because most values are at the mode, the mode is where a probability function is maximum. The mode occurs with the maximum likelihood. It is awkward to contemplate the mode for floating-point values where the probability is minuscule (and irrelevant) that any two values are identical. A more natural concept is to think of the mode as the bin containing the most values.

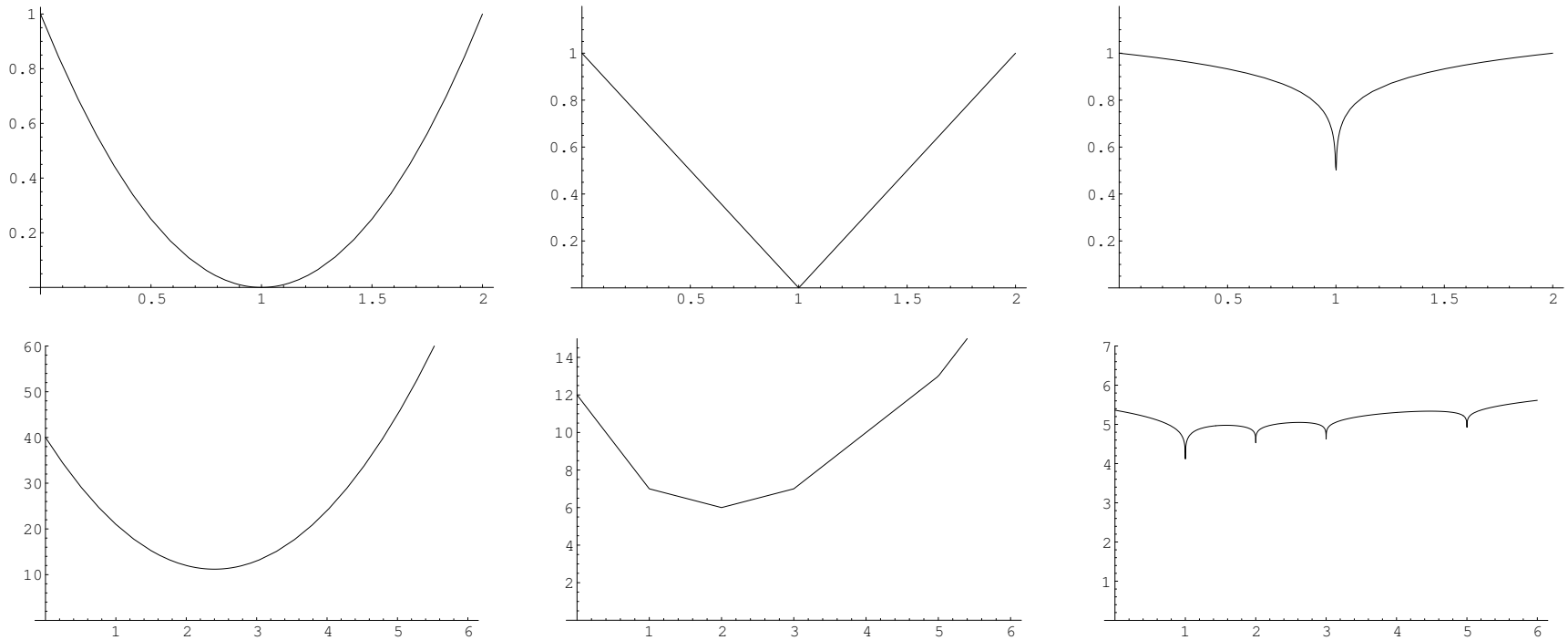


Figure 7.1: Mean, median, and mode. The coordinate is m. Top is the ℓ^2, ℓ^1, and $\ell^{1/10} \approx \ell^0$ measures of $m - 1$. Bottom is the same measures of the data set $(1, 1, 2, 3, 5)$. (Made with Mathematica.) noiz-norms [CR]

7.1.1. Percentiles and Hoare's algorithm

The median is the 50-th percentile. After residuals are ordered from smallest to largest, the 90-th percentile is the value with 10% of the values above and 90% below. At SEP the default value for clipping plots of field data is at the 98th percentile. In other words, magnitudes above the 98-th percentile are plotted at the 98-th percentile. Any percentile is most easily defined if the population of values a_i, for $i = 1, 2, ..., n$ has been sorted into order so that $a_i \leq a_{i+1}$ for all i. Then the 90-th percentile is a_k where $k = (90n)/100$.

We can save much work by using Hoare's algorithm which does not fully order the whole list, only enough of it to find the desired quantile. Hoare's algorithm is an outstanding example of the power of a recursive function, a function that calls itself. The main idea is this: We start by selecting a random value taken from the list of numbers. Then we split the list into two piles, one pile all values greater than the selected, the other pile all less. The quantile is in one of these piles, and by looking at their sizes, we know which one. So we repeat the process on that pile and ignore the other other one. Eventually the pile size reduces to one, and we have the answer.

In Fortran 77 or C it would be natural to split the list into two piles as follows:

We divide the list of numbers into two groups, a group below

a_k and another group above a_k. This reordering can be done "in place." Start one pointer at the top of the list and another at the bottom. Grab an arbitrary value from the list (such as the current value of a_k). March the two pointers towards each other until you have an upper value out of order with a_k and a lower value out of order with a_k. Swap the upper and lower value. Continue until the pointers merge somewhere midlist. Now you have divided the list into two sublists, one above your (random) value a_k and the other below.

Fortran 90 has some higher level intrinsic vector functions that simplify matters. When `a` is a vector and `ak` is a value, `a>ak` is a vector of logical values that are true for each component that is larger than `ak`. The integer count of how many components of `a` are larger than `ak` is given by the Fortran intrinsic function `count(a>ak)`. A vector containing only values less than `ak` is given by the Fortran intrinsic function `pack(a,a<ak)`.

Theoretically about $2n$ comparisons are expected to find the median of a list of n values. The code below (from Sergey Fomel) for this task is `quantile`. quantile

```
module quantile_mod {    # quantile finder.    median = quantile( size(a)/2, a)
contains
  recursive function quantile( k, a) result( value) {
    integer,              intent (in)  :: k          # position in array
    real, dimension (:), intent (in)  :: a
    real                               :: value      # output value of quantile
    integer                            :: j
    real                               :: ak
    ak = a( k)
    j = count( a < ak)                                # how many a(:) < ak
    if( j >= k)
                value = quantile( k, pack( a, a < ak))
    else {
        j = count( a > ak) + k - size( a)
        if( j > 0)
                value = quantile( j, pack( a, a > ak))
        else
                value = ak
        }
    }
}
```

Back

7.1.2. The weighted mean

The weighted mean m is

$$m = \frac{\sum_{i=1}^{N} w_i^2 d_i}{\sum_{i=1}^{N} w_i^2} \tag{7.7}$$

where $w_i^2 > 0$ is the squared weighting function. This is the solution to the ℓ^2 fitting problem $0 \approx w_i(m - d_i)$; in other words,

$$0 = \frac{d}{dm} \sum_{i=1}^{N} [w_i(m - d_i)]^2 \tag{7.8}$$

7.1.3. Weighted L.S. conjugate-direction template

The pseudocode for minimizing the *weighted* residual $\mathbf{0} \approx \mathbf{r} = \mathbf{W}(\mathbf{Fm} - \mathbf{d})$ by conjugate-direction method, is effectively like that for the unweighted method except that the initial residual is weighted and the operator $\mathbf{F}$ has the premultiplier $\mathbf{W}$. Naturally, the adjoint operator $\mathbf{F}'$ has the postmultiplier $\mathbf{W}'$. In some applications the weighting operator $\mathbf{W}$ is simply a weighting function or diagonal matrix (so then

$\mathbf{W} = \mathbf{W}'$) and in other applications, the weighting operator $\mathbf{W}$ may be an operator, like the derivative along a data recording trajectory (so then $\mathbf{W} \neq \mathbf{W}'$).

$\mathbf{r} \quad \longleftarrow \quad \mathbf{W(Fm - d)}$
iterate {
$\quad \Delta\mathbf{m} \quad \longleftarrow \quad \mathbf{F'W'\ r}$
$\quad \Delta\mathbf{r} \quad \longleftarrow \quad \mathbf{WF\ \Delta m}$
$\quad (\mathbf{m}, \mathbf{r}) \quad \longleftarrow \quad \text{cgstep}(\mathbf{m}, \mathbf{r}, \Delta\mathbf{m}, \Delta\mathbf{r})$
}

7.1.4. Multivariate ℓ^1 estimation by iterated reweighting

The easiest method of model fitting is linear least squares. This means minimizing the sums of squares of residuals (ℓ^2). On the other hand, we often encounter huge noises and it is much safer to minimize the sums of absolute values of residuals (ℓ^1). (The problem with ℓ^0 is that there are multiple minima, so the gradient is not a sensible way to seek the deepest).

There exist specialized techniques for handling ℓ^1 multivariate fitting problems. They should work better than the simple iterative reweighting outlined here.

A penalty function that ranges from ℓ^2 to ℓ^1, depending on the constant $\bar{r}$ is

$$E(\mathbf{r}) \quad = \quad \sum_i \left(\sqrt{1 + r_i^2/\bar{r}^2} - 1 \right) \tag{7.9}$$

Where $r_i/\bar{r}$ is small, the terms in the sum amount to $r_i^2/2\bar{r}^2$ and where $r_i^2/\bar{r}^2$ is large, the terms in the sum amount to $|r_i/\bar{r}|$. We define the residual as

$$r_i \quad = \quad \sum_j F_{ij} m_j - d_i \tag{7.10}$$

We will need

$$\frac{\partial r_i}{\partial m_k} \quad = \quad \sum_j F_{ij} \delta_{jk} \quad = \quad F_{ik} \tag{7.11}$$

where we briefly used the notation that δ_{jk} is 1 when $j = k$ and zero otherwise. Now, to let us find the descent direction $\Delta\mathbf{m}$, we will compute the k-th component

g_k of the gradient **g**. We have

$$g_k \quad = \quad \frac{\partial E}{\partial m_k} \quad = \quad \sum_i \frac{1}{\sqrt{1+r_i^2/\bar{r}^2}} \frac{r_i}{\bar{r}^2} \frac{\partial r_i}{\partial m_k} \tag{7.12}$$

$$\mathbf{g} \quad = \quad \Delta\mathbf{m} \quad = \quad \mathbf{F}' \, \mathbf{diag} \left(\frac{1}{\sqrt{1+r_i^2/\bar{r}^2}} \right) \mathbf{r} \tag{7.13}$$

where we have use the notation **diag**() to designate a diagonal matrix with its argument distributed along the diagonal.

Continuing, we notice that the new weighting of residuals has nothing to do with the linear relation between model perturbation and residual perturbation; that is, we retain the familiar relations, $\mathbf{r} = \mathbf{Fm} - \mathbf{d}$ and $\Delta\mathbf{r} = \mathbf{F}\Delta\mathbf{m}$.

In practice we have the question of how to choose $\bar{r}$. I suggest that $\bar{r}$ be proportional to median($|r_i|$) or some other percentile.

7.1.5. Nonlinear L.S. conjugate-direction template

Nonlinear optimization arises from two causes:

1. Nonlinear physics. The operator depends upon the solution being attained.
2. Nonlinear statistics. We need robust estimators like the ℓ^1 norm.

The computing template below is useful in both cases. It is almost the same as the template for weighted linear least-squares except that the residual is recomputed at each iteration. Starting from the usual weighted least-squares template we simply move the iteration statement a bit earlier.

iterate {

$$
\begin{aligned}
\mathbf{r} &\longleftarrow \mathbf{F m - d} \\
\mathbf{W} &\longleftarrow \mathbf{diag}[w(\mathbf{r})] \\
\mathbf{r} &\longleftarrow \mathbf{W r} \\
\Delta\mathbf{m} &\longleftarrow \mathbf{F}'\mathbf{W}'\,\mathbf{r} \\
\Delta\mathbf{r} &\longleftarrow \mathbf{W F}\,\Delta\mathbf{m} \\
(\mathbf{m},\mathbf{r}) &\longleftarrow \text{cgstep}(\mathbf{m},\mathbf{r},\Delta\mathbf{m},\Delta\mathbf{r})
\end{aligned}
$$

}

where **diag**$[w(\mathbf{r})]$ is whatever weighting function we choose along the diagonal of a diagonal matrix.

Now let us see how the weighting functions relate to robust estimation: Notice in the code template that **W** is applied twice in the definition of $\Delta\mathbf{m}$. Thus **W** is the square root of the diagonal operator in equation (7.13).

$$\mathbf{W} \quad = \quad \mathbf{diag} \left(\frac{1}{\sqrt{\sqrt{1 + r_i^2 / \overline{r}^2}}} \right) \tag{7.14}$$

Module `solver_irls` /prog:solver_irls implements the computational template above. In addition to the usual set of arguments from the `solver()` subroutine /prog:smallsolver, it accepts a user-defined function (parameter `wght`) for computing residual weights. Parameters `nmem` and `nfreq` control the restarting schedule of the iterative scheme. solver_irls

We can ask whether `cgstep()`, which was not designed with nonlinear least-squares in mind, is doing the right thing with the weighting function. First, we know we are doing weighted linear least-squares correctly. Then we recall that on

```
module solver_irls_mod {                          # 0 = Wdiag Wop (Fop Jop m -
d) and periodic restart
  use chain0_mod + solver_report_mod
  logical, parameter, private  :: T = .true., F = .false.
contains
  subroutine solver_irls( m,d, Fop, stepper, niter &
  ,             Wop,Jop,Wdiag,m0,nmem,nfreq,err,resd,mmov,rmov,verb) {
    optional :: Wop,Jop,Wdiag,m0,nmem,nfreq,err,resd,mmov,rmov,verb
    interface { #-------------------------- begin definitions -----------
       integer function Fop  (adj,add,m,d){real::m(:),d(:);logical,intent(in)::adj,add}
       integer function Wop  (adj,add,m,d){real::m(:),d(:);logical,intent(in)::adj,add}
       integer function Jop  (adj,add,m,d){real::m(:),d(:);logical,intent(in)::adj,add}
       integer function Wdiag(res, w)     {real::res(:),w(:)}
       integer function stepper(forget,m,g,rr,gg) {
         real, dimension(:)  ::         m,g,rr,gg
         logical             :: forget            }
    }
    real, dimension(:),    intent(in)          :: d, m0
    integer,               intent(in)          :: niter, nmem, nfreq
    logical,               intent(in)          :: verb
    real, dimension(:),    intent(out)         :: m,err, resd
    real, dimension(:,:),  intent(out)         ::        rmov, mmov
    real, dimension(size(m))                   :: g
    real, dimension(size(d)),          target  :: rr, gg
    real, dimension(size(d)+size(m)), target   :: tt
    real, dimension(:), pointer                :: rd, gd, td
    real, dimension(:), pointer                :: rm, gm, tm, wht
    integer                                    :: iter, stat
    logical                                    :: forget
    rd => rr(1:size(d));
    gd => gg(1:size(d));
    td => tt(1:size(d)); tm => tt(1+size(d):)
    if(present( Wop)) stat=Wop(F,F,-d,rd) # begin initialization --------
    else rd = -d                                          #Rd = -W  d
    if(present(Wdiag))allocate(wht(size(d)))
    if(present( m0)){ m=m0                                #m  =     m0
        if(present( Wop)) call chain0(Wop,Fop,F,T,m,rd,td)
        else              stat =          Fop(F,T,m,rd   ) #Rd+=  WF m0
```

the first iteration, the conjugate-directions technique reduces to steepest descent, which amounts to a calculation of the scale factor α with

$$\alpha \quad = \quad - \frac{\Delta \mathbf{r} \cdot \mathbf{r}}{\Delta \mathbf{r} \cdot \Delta \mathbf{r}} \tag{7.15}$$

Of course, `cgstep()` knows nothing about the weighting function, but notice that the iteration loop above nicely inserts the weighting function both in $\mathbf{r}$ and in $\Delta\mathbf{r}$, as required by (7.15).

Experience shows that difficulties arise when the weighting function varies rapidly from one iteration to the next. Naturally, the conjugate-direction method, which remembers the previous iteration, will have an inappropriate memory if the weighting function changes too rapidly. A practical approach is to be sure the changes in the weighting function are slowly variable.

7.1.6. Minimizing the Cauchy function

A good trick (I discovered accidentally) is to use the weight

$$\mathbf{W} \quad = \quad \mathbf{diag}\left(\frac{1}{\sqrt{1+r_i^2/\bar{r}^2}}\right) \tag{7.16}$$

Sergey Fomel points out that this weight arises from minimizing the Cauchy function:

$$E(\mathbf{r}) \quad = \quad \sum_i \log(1+r_i^2/\bar{r}^2) \tag{7.17}$$

A plot of this function is found in Figure 7.2.

Because the second derivative is not positive everywhere, the Cauchy function introduces the possibility of multiple solutions, but because of the good results we see in Figure 7.3, you might like to try it anyway. Perhaps the reason it seems to work so well is that it uses mostly residuals of "average size," not the big ones or the small ones. This happens because $\Delta\mathbf{m}$ is made from $\mathbf{F}'$ and the components of

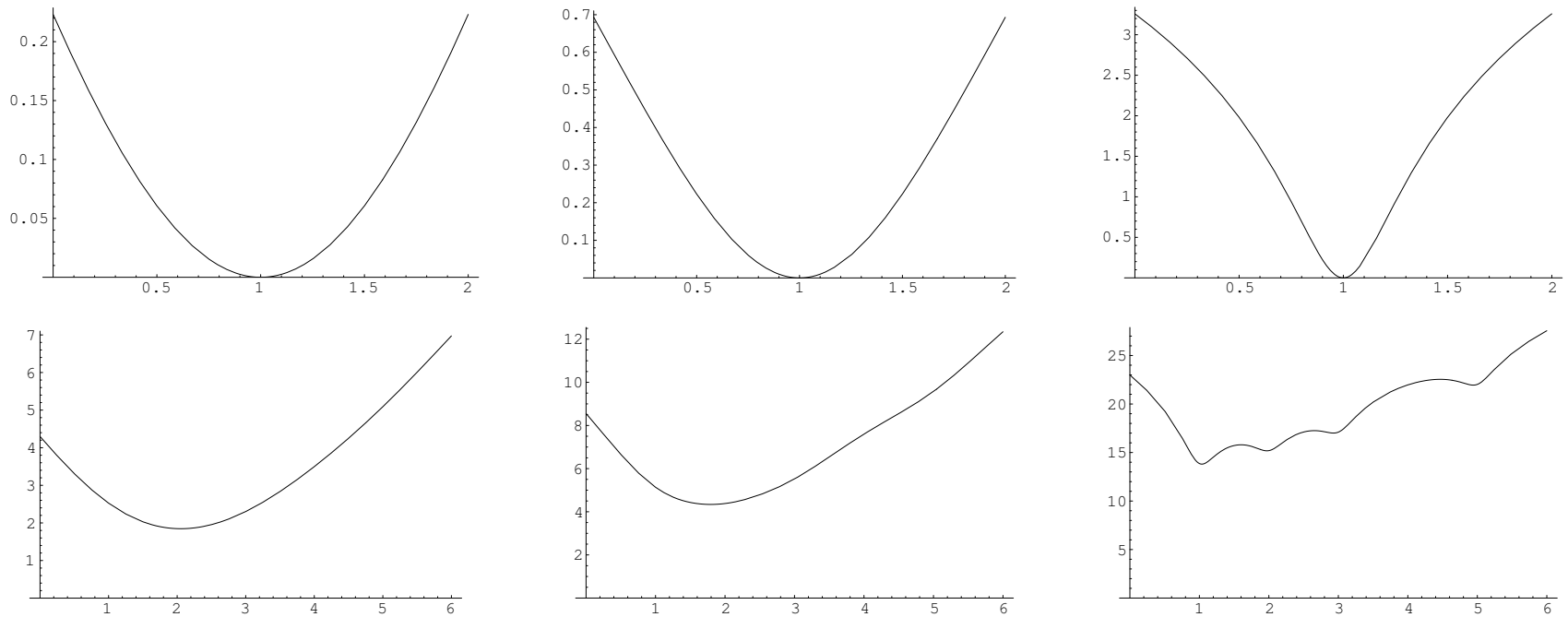


Figure 7.2: The coordinate is m. Top is Cauchy measures of $m-1$. Bottom is the same measures of the data set $(1,1,2,3,5)$. Left, center, and right are for $\bar{r} = (2,1,.2)$. noiz-cauchy [CR]

$\mathbf{W}^2\mathbf{r}$ which are a function $r_i/(1+r_i^2/\bar{r}^2)$ that is maximum for those residuals near $\bar{r}$.

Module `irls` /prog:irls supplies two useful weighting functions that can be interchanged as arguments to the reweighted scheme /prog:solver_irls. irls

7.2. NOISE BURSTS

Sometimes noise comes in isolated spikes. Sometimes it comes in bursts or bunches (like grapes). Figure 7.3 is a simple one-dimensional example of a periodic signal plus spikes and bursts. Three processes are applied to this data, despike and two flavors of deburst. Here we will examine the processes used. (For even better results, see Figure 7.5.)

7.2.1. De-spiking with median smoothing

The easiest method to remove spikes is to pass a moving window across the data and output the median value in the window. This method of despiking was done in Figure 7.3, which shows a problem of the method: The window is not long

```
module irls {
  use quantile_mod
contains
  integer function l1 (res, weight)  {
    real, dimension (:)  :: res, weight
    real                 :: rbar
    rbar = quantile( int( 0.5*size(res)), abs (res))            # median
    weight = 1. / sqrt( sqrt (1. + (res/rbar)**2));   l1 = 0
    }
  integer function cauchy (res, weight)  {
    real, dimension (:)  :: res, weight
    real                 :: rbar
    rbar = quantile( int( 0.5*size(res)), abs (res))             # median
    weight = 1. / sqrt (1. + (res/rbar)**2);          cauchy = 0
    }
}
```

Back

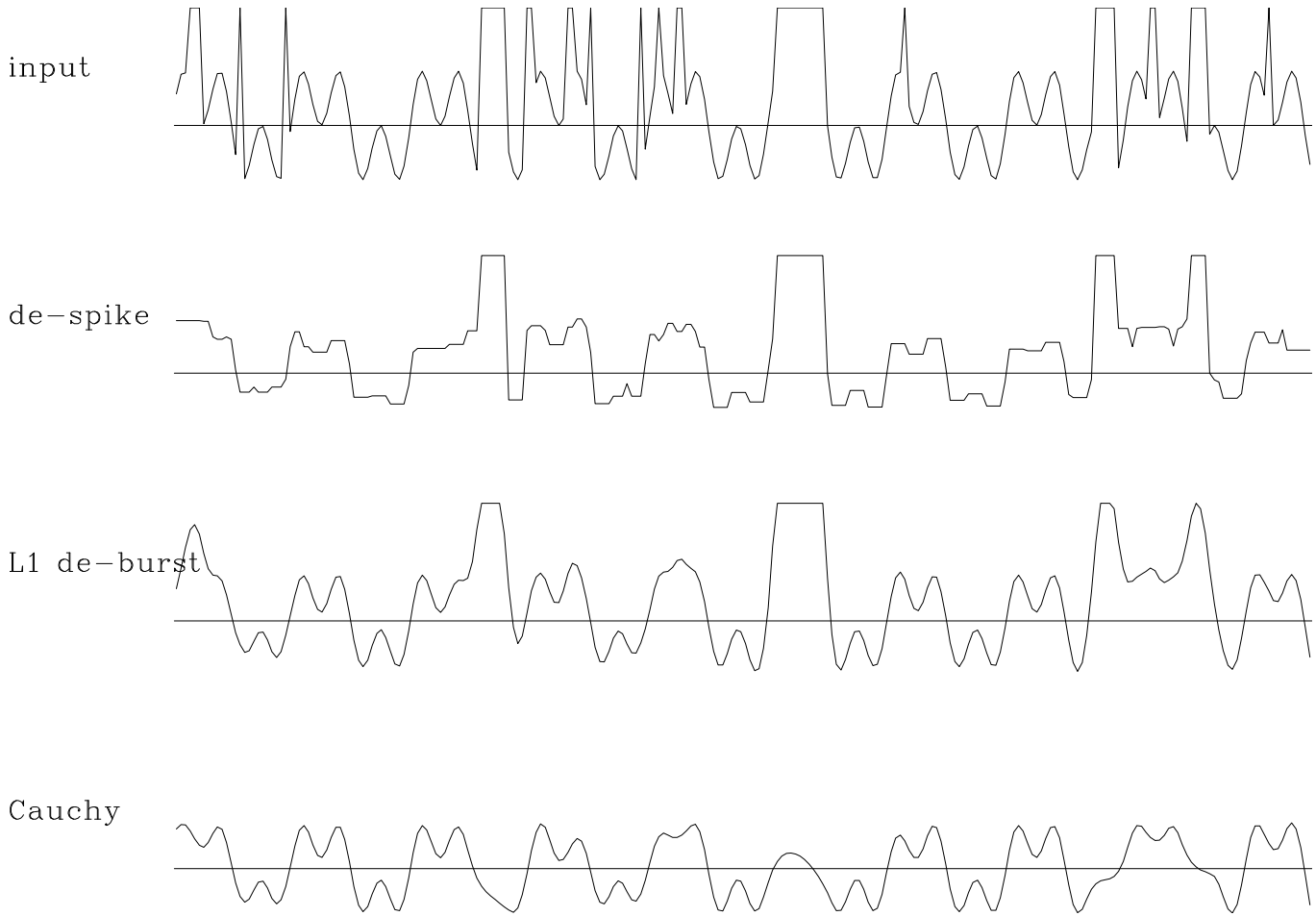


Figure 7.3: Top is synthetic data with noise spikes and bursts. (Most bursts are a hundred times larger than shown.) Next is after running medians. Bottom is after the two processes described here. noiz-burst90 [ER]

enough to clean the long bursts, but it is already so long that it distorts the signal by flattening its peaks. The window size really should not be chosen in advance but should depend upon by what is encountered on the data. This I have not done because the long-burst problem is solved by another method described next.

7.2.2. De-bursting

Most signals are smooth, but running medians assume they have no curvature. An alternate expression of this assumption is that the signal has minimal curvature $0 \approx h_{i+1} - 2h_i + h_{i-1}$; in other words, $\mathbf{0} \approx \nabla^2 \mathbf{h}$. Thus we propose to create the cleaned-up data $\mathbf{h}$ from the observed data $\mathbf{d}$ with the fitting problem

$$\begin{aligned} 0 &\approx \mathbf{W}(\mathbf{h} - \mathbf{d}) \\ 0 &\approx \epsilon \, \nabla^2 \mathbf{h} \end{aligned} \tag{7.18}$$

where $\mathbf{W}$ is a diagonal matrix with weights sprinkled along the diagonal, and where ∇^2 is a matrix with a roughener like $(1, -2, 1)$ distributed along the diagonal. This is shown in Figure 7.3 with $\epsilon = 1$. Experience showed similar performances for $0 \approx \nabla \mathbf{h}$ and $0 \approx \nabla^2 \mathbf{h}$. Better results, however, will be found later in Figure 7.5, where

the ∇^2 operator is replaced by an operator designed to predict this very predictable signal.

7.3. MEDIAN BINNING

We usually add data into bins. When the data has erratic noise, we might prefer to take the median of the values in each bin. Subroutine `medianbin2()` (in the library, but not listed here) performs the chore. It is a little tricky because we first need to find out how many data values go into each bin, then we must allocate that space and copy each data value from its track location to its bin location. Finally we take the median in the bin. A small annoyance with medians is that when bins have an even number of points, like two, there no middle. To handle this problem, subroutine `medianbin2()` uses the average of the middle two points.

A useful byproduct of the calculation is the residual: For each data point its bin median is subtracted. The residual can be used to remove suspicious points before any traditional least-squares analysis is made. An overall strategy could be this: First a coarse binning with many points per bin, to identify suspicious data values, which are set aside. Then a sophisticated least squares analysis leading to a high-

resolution depth model. If our search target is small, recalculate the residual with the high-resolution model and reexamine the suspicious data values.

Figure 7.4 compares the water depth in the Sea of Galilee with and without median binning. The difference does not seem great here but it is more significant than it looks. Later processing will distinguish between empty bins (containing an exact zero) and bins with small values in them. Because of the way the depth sounder works, it often records an erroneously near-zero depth. This will make a mess of our later processing (missing data fill) unless we cast out those data values. This was done by median binning in Figure 7.4 but the change is disguised by the many empty bins.

Median binning is a useful tool, but where bins are so small that they hold only one or two points, there the median for the bin is the same as the usual arithmetic average.

7.4. ROW NORMALIZED PEF

We often run into bursty noise. This can overwhelm the estimate of a prediction-error filter. To overcome this problem we can use a weighting function. The weight

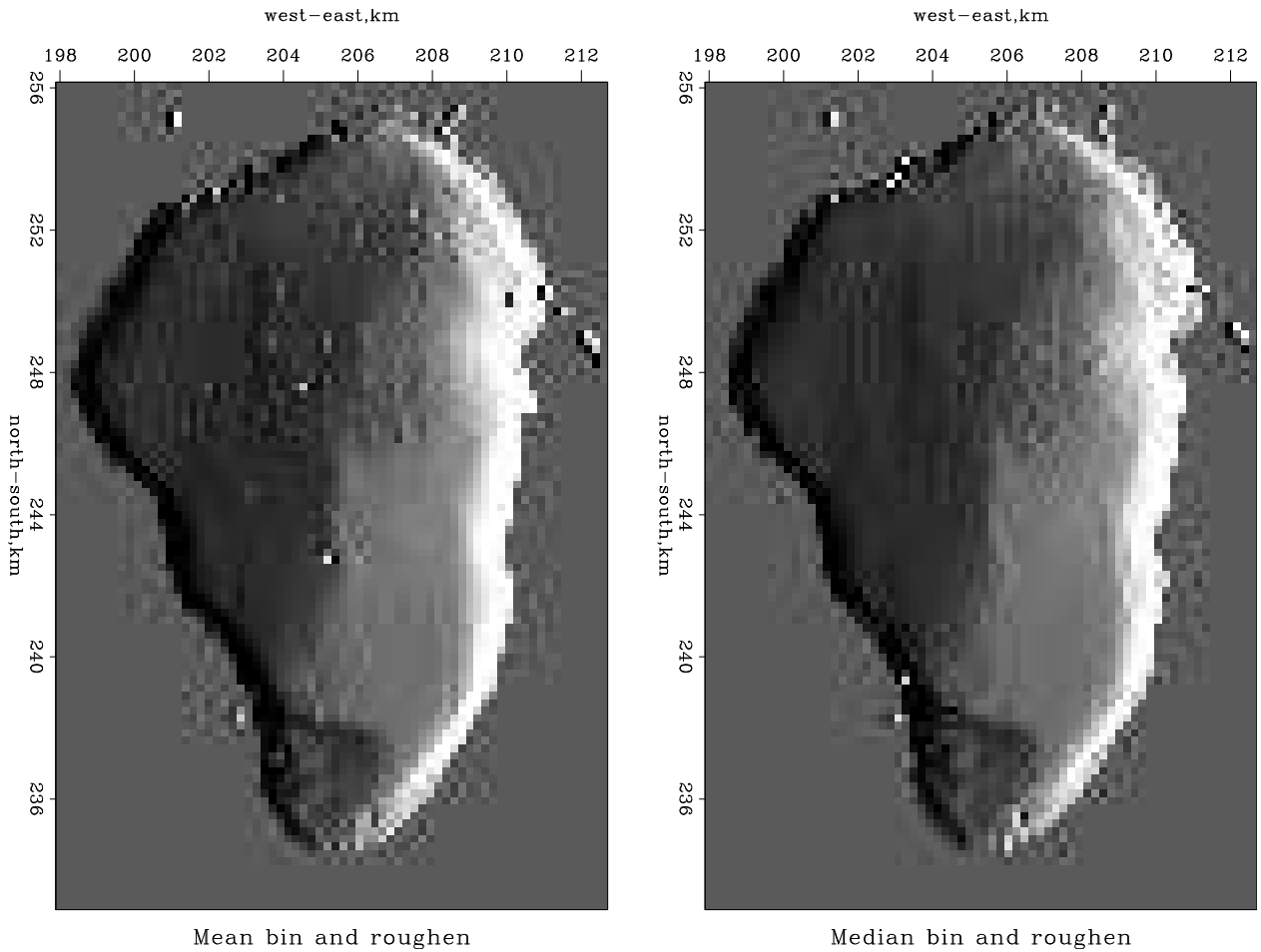


Figure 7.4: Galilee water depth binned and roughened. Left is binning with the mean, right with the median. noiz-medbin90 [ER,M]

for each row in fitting matrix (6.27) is adjusted so that each row has about the same contribution as each other row. A first idea is that the weight for the n-th row would be the inverse of the sum of the absolute values of the row. This is easy to compute: First make a vector the size of the PEF **a** but with each element unity. Second, take a copy of the signal vector **y** but with the absolute value of each component. Third, convolve the two. The convolution of the ones with the absolute values could be the inverse of the weighting function we seek. However, any time we are forming an inverse we need to think about the possibility of dividing by zero, how it could arise, and how divisions by "near zero" could be even worse (because a poor result is not immediately recognized). Perhaps we should use something between ℓ^1 and ℓ^2 or Cauchy. In any case, we must choose a scaling parameter that separates "average" rows from unusually large ones. For this choice in subroutine `rnpef1()`, I chose the median.

7.5. DEBURST

We can use the same technique to throw out fitting equations from defective data that we use for missing data. Recall the theory and discussion leading up to Figure

7.3. There we identified defective data by its lack of continuity. We used the fitting equations $0 \approx w_i(y_{i+1} - 2y_i + y_{i-1})$ where the weights w_i were chosen to be approximately the inverse to the residual $(y_{i+1} - 2y_i + y_{i-1})$ itself.

Here we will first use the second derivative (Laplacian in 1-D) to throw out bad points, while we determine the PEF. Having the PEF, we use it to fill in the missing data. pefest The result of this "PEF-deburst" processing is shown in Figure 7.5.

Given the PEF that comes out of `pefest1()`[1], subroutine `fixbad1()` below convolves it with the data and looks for anomalous large outputs. For each that is found, the input data is declared defective and set to zero. Then subroutine `mis1()` /prog:mis2 is invoked to replace the zeroed values by reasonable ones. fixbad

7.5.1. Potential seismic applications of two-stage infill

Two-stage data infill has many applications that I have hardly begun to investigate.

---

[1] If you are losing track of subroutines defined earlier, look at the top of the module to see what other modules it `uses`. Then look in the index to find page numbers of those modules.

```
module pefest {      # Estimate a PEF avoiding zeros and bursty noise on input.
  use quantile_mod
  use helicon
  use misinput
  use pef
contains
  subroutine pefest1( niter, yy, aa) {
    integer, intent( in)         :: niter
    real, dimension( :), pointer :: yy
    type( filter)                :: aa
    real, dimension(:),allocatable  :: rr
    real                         :: rbar
    integer                      :: stat
                allocate(rr(size(yy)))
    call helicon_init( aa)                              # starting guess
    stat = helicon_lop( .false., .false., yy, rr)
    rbar = quantile( size( yy)/3, abs( rr))             # rbar=(r safe below rbar)
    where( aa%mis)  yy = 0.
    call find_mask(( yy /= 0. .and. abs( rr) < 5 * rbar), aa)
    call find_pef( yy, aa, niter)
                deallocate(rr)
  }
}
```

Back

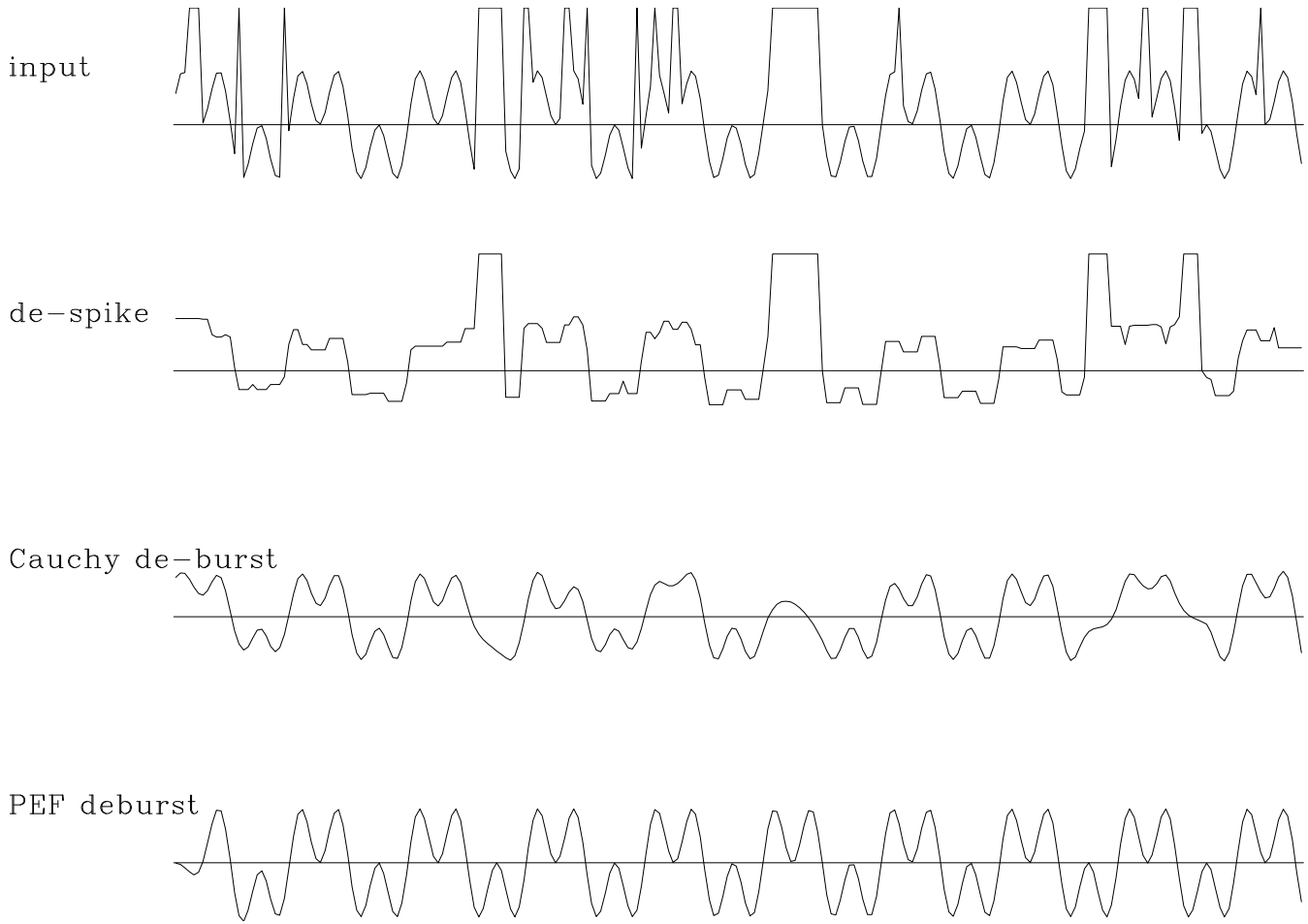


Figure 7.5: Top is synthetic data with noise spikes and bursts. (Some bursts are fifty times larger than shown.) Next is after running medians. Next is Laplacian filter Cauchy deburst processing. Last is PEF-deburst processing. noiz-pefdeburst90 [ER]

```
module fixbad { # Given a PEF, find bad data and restore it.
  use mis2_mod
  use helicon
  use quantile_mod
contains
  subroutine fixbad1 (niter, aa, yy) {
    integer,       intent (in)      :: niter
    type( filter), intent (in)      :: aa
    real,    dimension (:)          :: yy
    real,    dimension (size (yy))  :: rr
    logical, dimension (size (yy))  :: known
    integer                         :: stat
    call helicon_init( aa)
    stat = helicon_lop (.false., .false., yy, rr); rr = abs (rr)
    known = ( yy > 0.) .and. ( rr < 4. * quantile( size(rr)/2, rr))
    call mis2 (niter, yy, aa, known, 2)
 }
}
```

Back

Shot continuation is an obvious task for a data-cube extrapolation program. There are two applications of shot-continuation. First is the obvious one of repairing holes in data in an unobtrusive way. Second is to cooperate with reflection tomographic studies such as that proposed by Matthias Schwab.

Offset continuation is a well-developed topic because of its close link with dip moveout (DMO). DMO is heavily used in the industry. I do not know how the data-cube extrapolation code I am designing here would fit into DMO and stacking, but because these are such important processes, the appearance of a fundamentally new tool like this should be of interest. It is curious that the DMO operator is traditionally derived from theory, and the theory requires the unknown velocity function of depth, whereas here I propose estimating the offset continuation operator directly from the data itself, without the need of a velocity model.

Obviously, one application is to extrapolate off the sides of a constant-offset section. This would reduce migration semicircles at the survey's ends.

Another application is to extrapolate off the cable ends of a common-midpoint gather or a common shot point gather. This could enhance the prediction of multiple reflections or reduce artifacts in velocity analysis.

Obviously, the methodology and code in this chapter is easily extendable to

four dimensions (prestack 3-D data).

7.6. TWO 1-D PEFS VERSUS ONE 2-D PEF

Here we look at the difference between using two 1-D PEFs, and one 2-D PEF. Figure 7.6 shows an example of sparse tracks; it is not realistic in the upper-left corner (where it will be used for testing), in a quarter-circular disk where the data covers the model densely. Such a dense region is ideal for determining the 2-D PEF. Indeed, we cannot determine a 2-D PEF from the sparse data lines, because at any place you put the filter (unless there are enough adjacent data lines), unknown filter coefficients will multiply missing data. So every fitting goal is nonlinear and hence abandoned by the algorithm.

The set of test data shown in Figure 7.6 is a superposition of three functions like plane waves. One plane wave looks like low-frequency horizontal layers. Notice that the various layers vary in strength with depth. The second wave is dipping about 30° down to the right and its waveform is perfectly sinusoidal. The third wave dips down 45° to the left and its waveform is bandpassed random noise like the horizontal beds. These waves will be handled differently by different processing

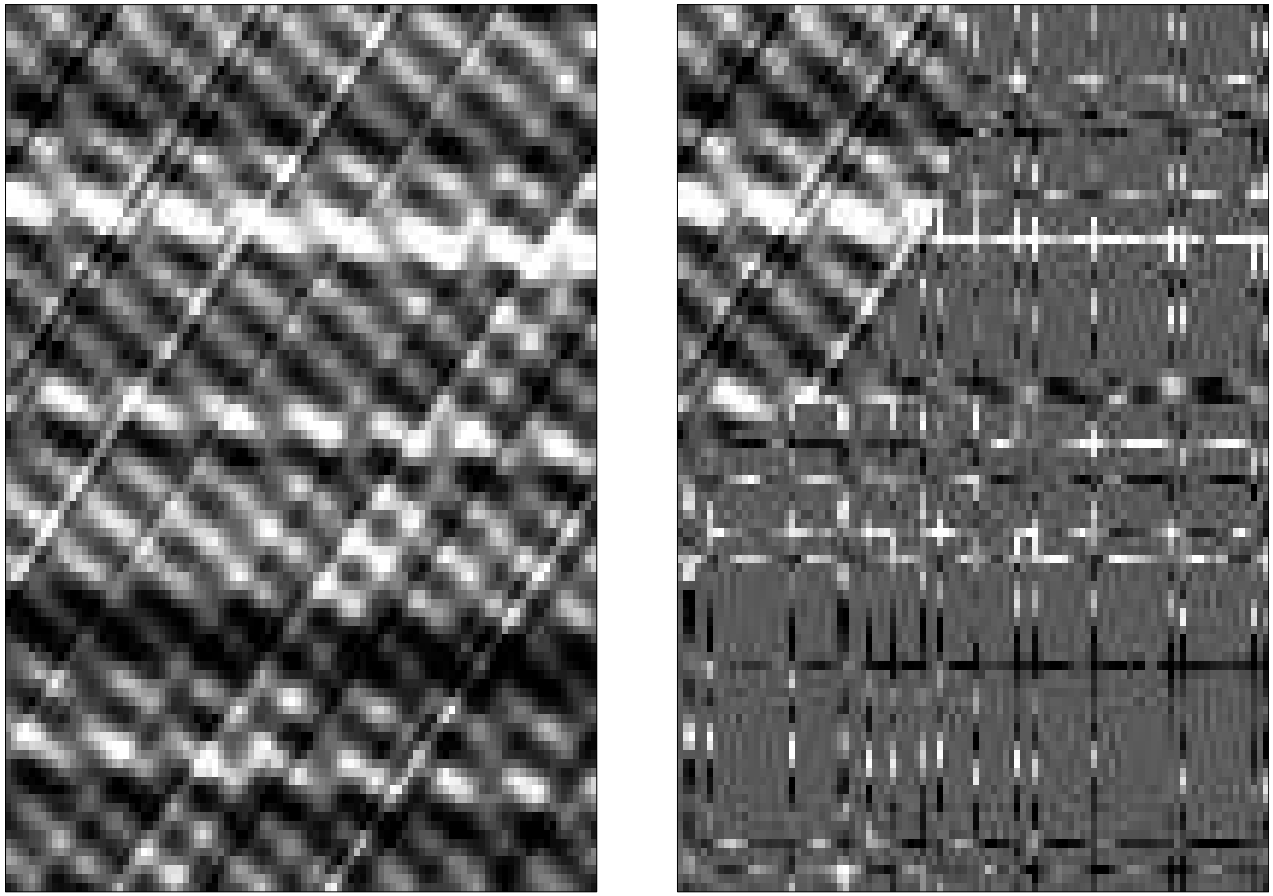


Figure 7.6: Synthetic wavefield (left) and as observed over survey lines (right). The wavefield is a superposition of waves from three directions. noiz-duelin90 [ER]

schemes, so I hope you can identify all three. If you have difficulty, view the figure at a grazing angle from various directions.

Later we will make use of the dense data region, but first let **U** be the east-west PE operator and **V** be the north-south operator and let the signal or image be $\mathbf{h} = h(x, y)$. The fitting residuals are

$$\begin{aligned} \mathbf{0} &\approx (\mathbf{I} - \mathbf{J})(\mathbf{h} - \mathbf{d}) \\ \mathbf{0} &\approx \mathbf{U}\,\mathbf{h} \\ \mathbf{0} &\approx \mathbf{V}\,\mathbf{h} \end{aligned} \tag{7.19}$$

where **d** is data (or binned data) and $(\mathbf{I} - \mathbf{J})$ masks the map onto the data.

Figure 7.7 shows the result of using a single one-dimensional PEF along either the vertical or the horizontal axis.

Figure 7.8 compares the use of a pair of 1-D PEFs versus a single 2-D PEF (which needs the "cheat" corner in Figure 7.6. Studying Figure 7.8 we conclude (what theory predicts) that

- These waves are predictable with a pair of 1-D filters:
  - Horizontal (or vertical) plane-wave with random waveform

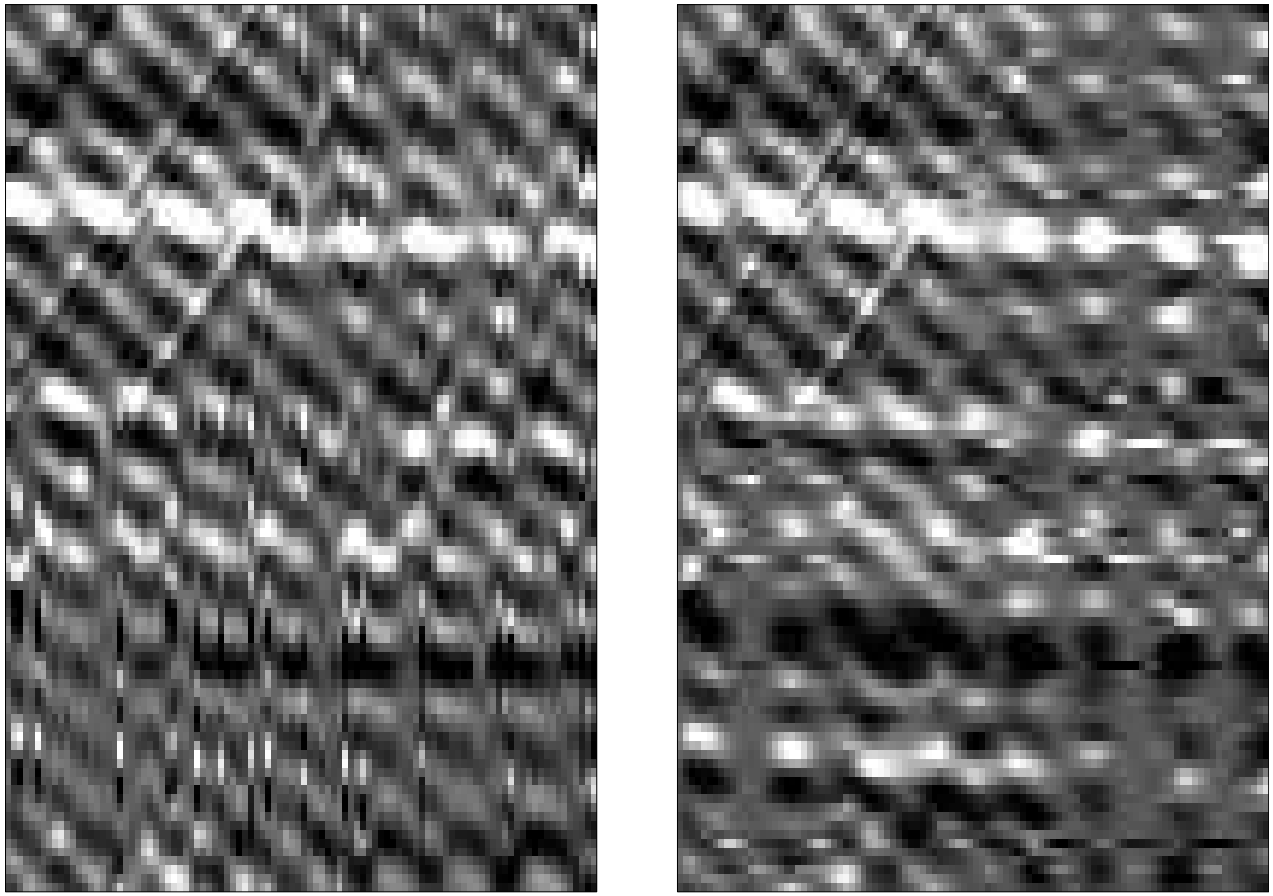


Figure 7.7: Interpolation by 1-D PEF along the vertical axis (left) and along the horizontal axis (right). noiz-dueleither90 [ER]

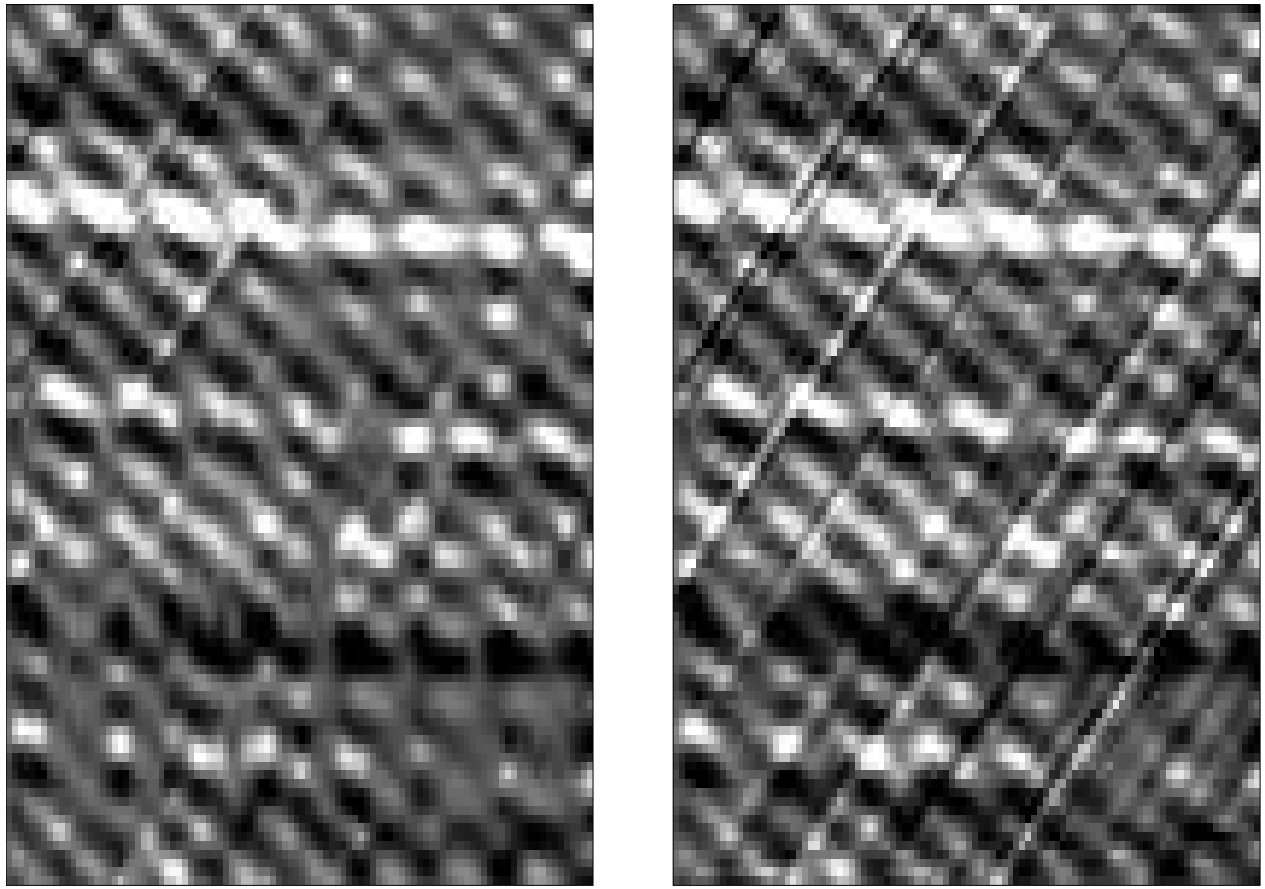


Figure 7.8: Data infilled by a pair of 1-D PEFs (left) and by a single 2-D PEF (right). noiz-duelversus90 [ER,M]

  - Dipping plane-wave with a sinusoidal waveform

- These waves are predictable with a single 2-D filter:
  - both of the above
  - Dipping plane-wave with a random waveform

7.7. ALTITUDE OF SEA SURFACE NEAR MADAGASCAR

A satellite points a radar at the ground and receives echos we investigate here. These echos are recorded only over the ocean. The echo tells the distance from the orbit to the ocean surface. After various corrections are made for earth and orbit ellipticities the residual shows tides, wind stress on the surface, and surprisingly a signal proportional to the depth of the water. Gravity of mountains on the water bottom pulls water towards them raising sea level there.

The raw data investigated here[2] had a strong north-south tilt which I[3] removed at the outset. Figure 7.9 gives our first view of altimetry data (ocean height) from southeast of the island of Madagascar. About all we can see is satellite tracks. The satellite is in a circular polar orbit. To us the sun seems to rotate east to west as does the circular satellite orbit. Consequently, when the satellite moves northward across the site we get altitude measurements along a SE-NW line. When it moves southward we get measurements along a NE-SW line. This data is from the cold war era. At that time dense data above the $-30°$ parallel was secret although sparse data was available. (The restriction had to do with precision guidance of missiles. Would the missile hit the silo? or miss it by enough to save the retaliation missile?)

Here are some definitions: Let components of **d** be the data, altitude measured along a satellite track. The model space is **h**, altitude in the (x, y)-plane. Let **L** denote the 2-D linear interpolation operator from the track to the plane. Let **H** be the helix derivative, a filter with response $\sqrt{k_x^2 + k_y^2}$. Except where otherwise noted,

---

[2] I wish to thank David T. Sandwell http://topex.ucsd.edu/ for providing me with this subset of satellite altimetry data, commonly known as Topex-Posidon data.

[3] The calculations here were all done for us by Jesse Lomask.

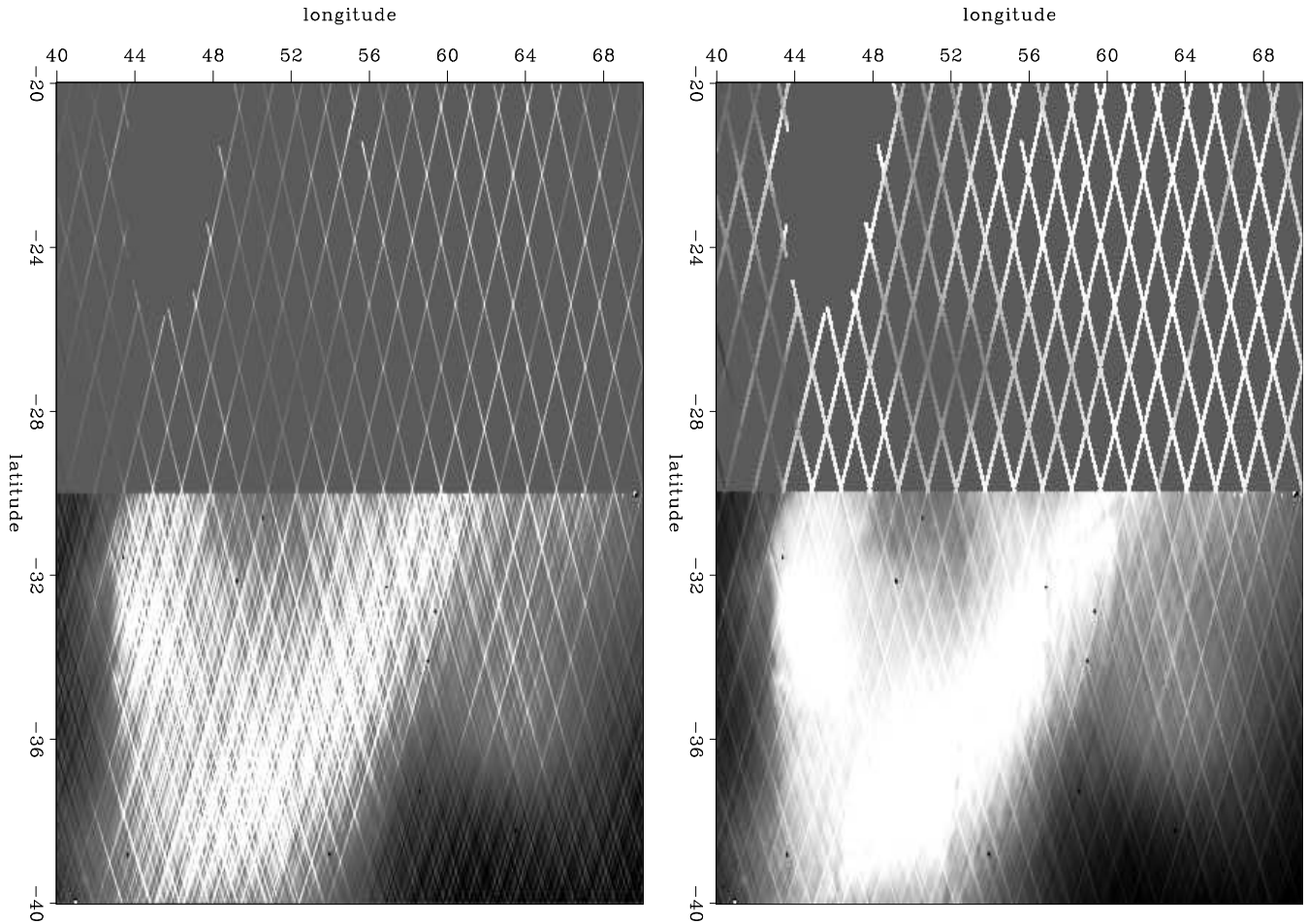


Figure 7.9: Sea height under satellite tracks. The island of Madagascar is in the empty area at $(46°, -22°)$. Left is the adjoint $\mathbf{L}'\mathbf{d}$. Right is the adjoint normalized by the bin count, $\mathbf{diag}(\mathbf{L}'\mathbf{1})^{-1}\mathbf{L}'\mathbf{d}$. You might notice a few huge, bad data values. Overall, the topographic function is too smooth, suggesting we need a roughener. noiz-jesse1 [ER,M]

the roughened image $\mathbf{p}$ is the preconditioned variable $\mathbf{p} = \mathbf{Hh}$. The derivative along a track in data space is $\frac{d}{dt}$. A weighting function that vanishes when any filter hits a track end or a bad data point is $\mathbf{W}$.

Figure 7.10 shows the entire data space, over a half million data points (actually 537974). Altitude is measured along many tracks across the image. In Figure 7.10 the tracks are placed end-to-end, so it is one long vector (displayed in about 50 signal rows). A vector of equal length is the missing data marker vector. This vector is filled with zeros everywhere except where data is missing or known bad or known to be at the ends of the tracks. The long tracks are the ones that are sparse in the north.

Figure 7.11 brings this information into model space. Applying the adjoint of the linear interpolation operator $\mathbf{L}'$ to the data $\mathbf{d}$ gave our first image $\mathbf{L}'\mathbf{d}$ in model space in Figure 7.9. The track noise was so large that roughening it made it worse. A more inviting image arose when I normalized the image before roughening it. Put a vector of all ones $\mathbf{1}$ into the adjoint of the linear interpolation operator $\mathbf{L}'$. What comes out $\mathbf{L}'\mathbf{1}$ is roughly the number of data points landing in each pixel in model space. More precisely, it is the sum of the linear interpolation weights. This then, if it is not zero, is used as a divisor. The division accounts for several tracks

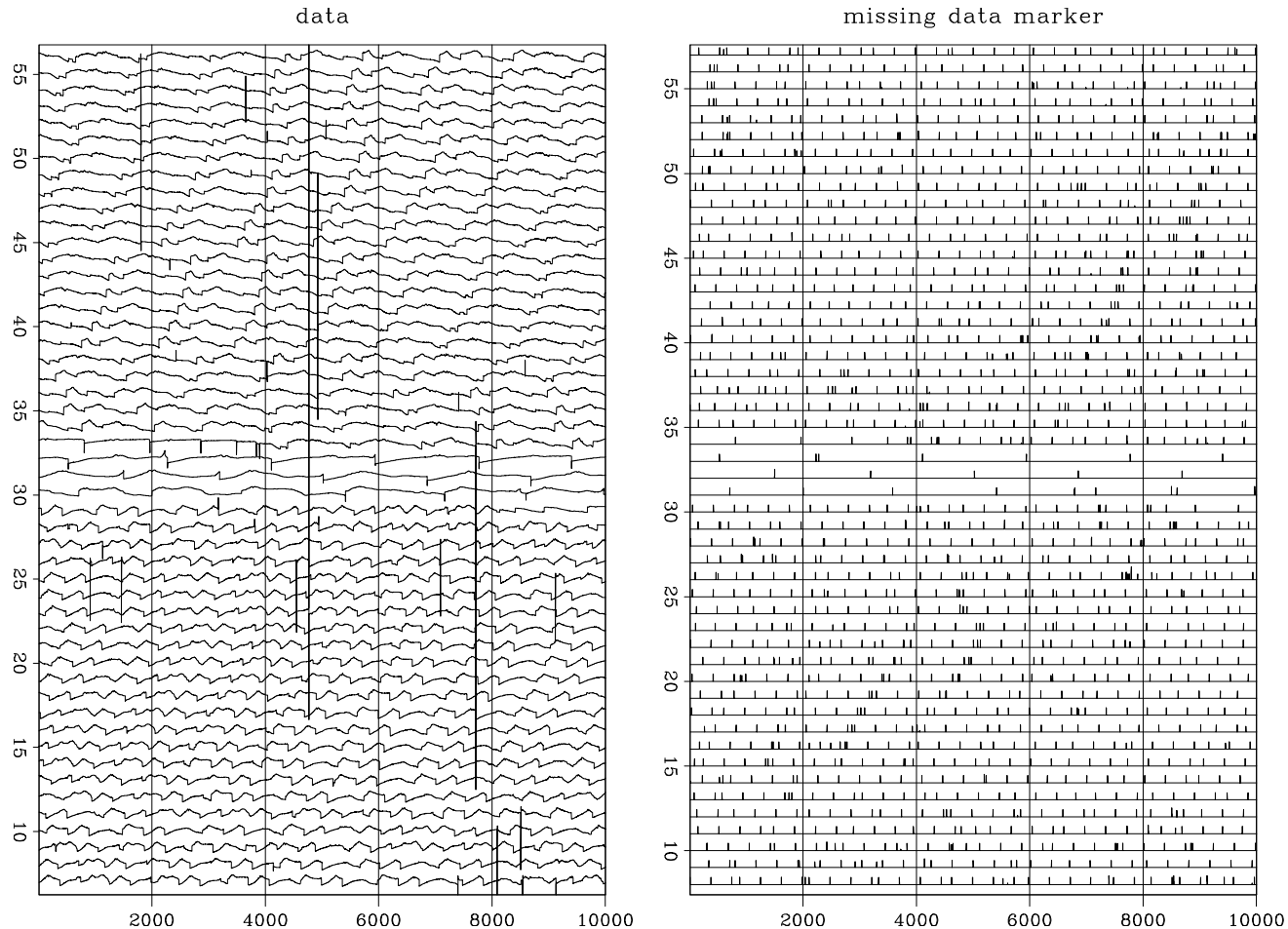


Figure 7.10: All the data **d** and the missing data markers. noiz-jesse5 [ER,M]

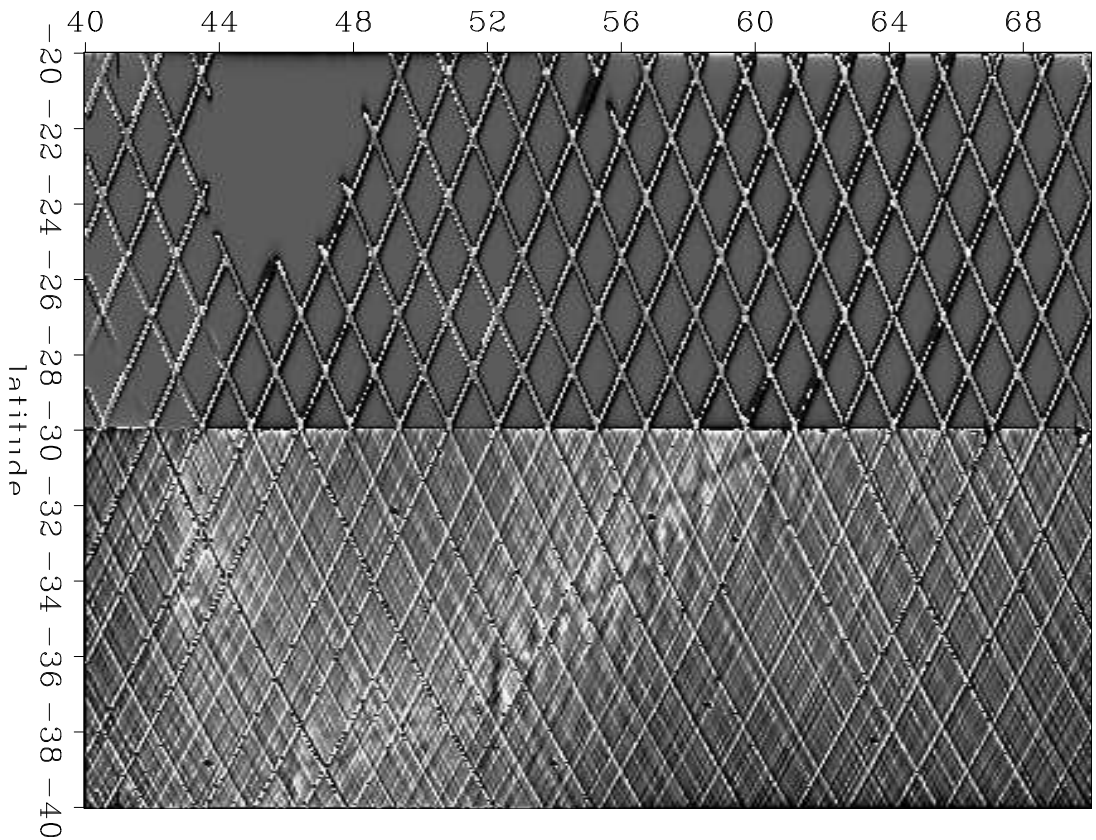


Figure 7.11: The roughened, normalized adjoint, $\mathbf{H}\ \mathbf{diag}(\mathbf{L}'\mathbf{1})^{-1}\mathbf{L}'\mathbf{d}$. Some topography is perceptible through a maze of tracks. noiz-jesse2 [ER,M]

contributing to one pixel. In matrix formalism this image is $\mathbf{diag}(\mathbf{L}'\mathbf{1})^{-1}\mathbf{L}'\mathbf{d}$. In Figure 7.11 this image is roughened with the helix derivative $\mathbf{H}$.

There is a simple way here to make a nice image—roughen along data tracks. This is done in Figure 7.12. The result is two attractive images, one for each track direction. Unfortunately, there is no simple relationship between the two images. We cannot simply add them because the shadows go in different directions. Notice also that each image has noticeable tracks that we would like to suppress further.

A geological side note: The strongest line, the line that marches along the image from southwest to northeast is a sea-floor spreading axis. Magma emerges along this line as a source growing plates that are spreading apart. Here the spreading is in the north-south direction. The many vertical lines in the image are called "transform faults".

Fortunately, we know how to merge the data. The basic trick is to form the track derivative not on the data (which would falsify it) but on the residual which (in Fourier space) can be understood as choosing a different weighting function for the statistics. A track derivative on the residual is actually two track derivatives, one on the observed data, the other on the modeled data. Both data sets are changed in the same way. Figure 7.13 shows the result. The altitude function remains too

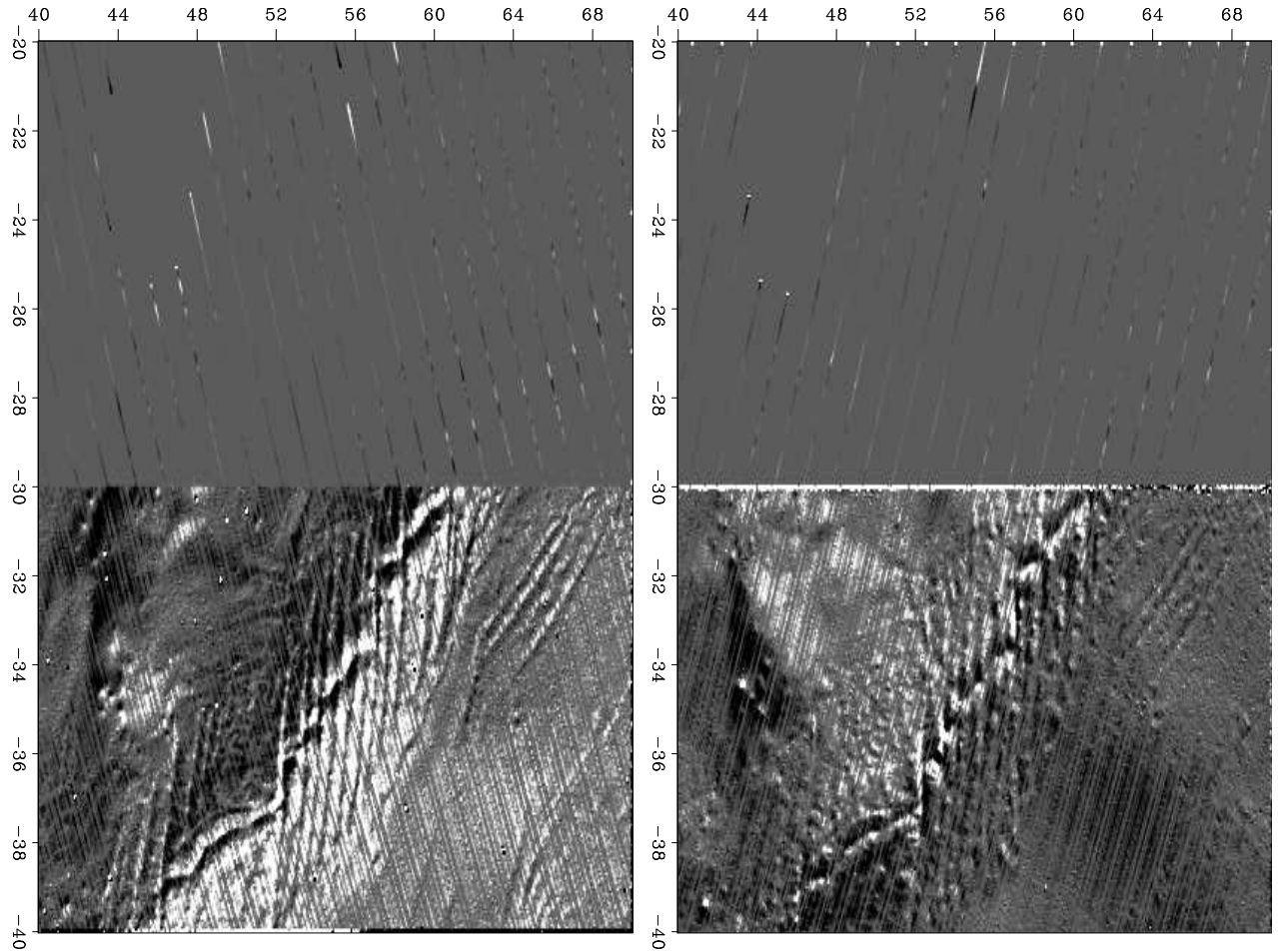


Figure 7.12: With a simple roughening derivative in data space, model space shows two nice topographic images. Let **n** denote ascending tracks. Let **s** denote descending tracks. Left is $\mathbf{L}' \frac{d}{dt}\mathbf{n}$. Right is $\mathbf{L}' \frac{d}{dt}\mathbf{s}$. noiz-jesse3 [ER,M]

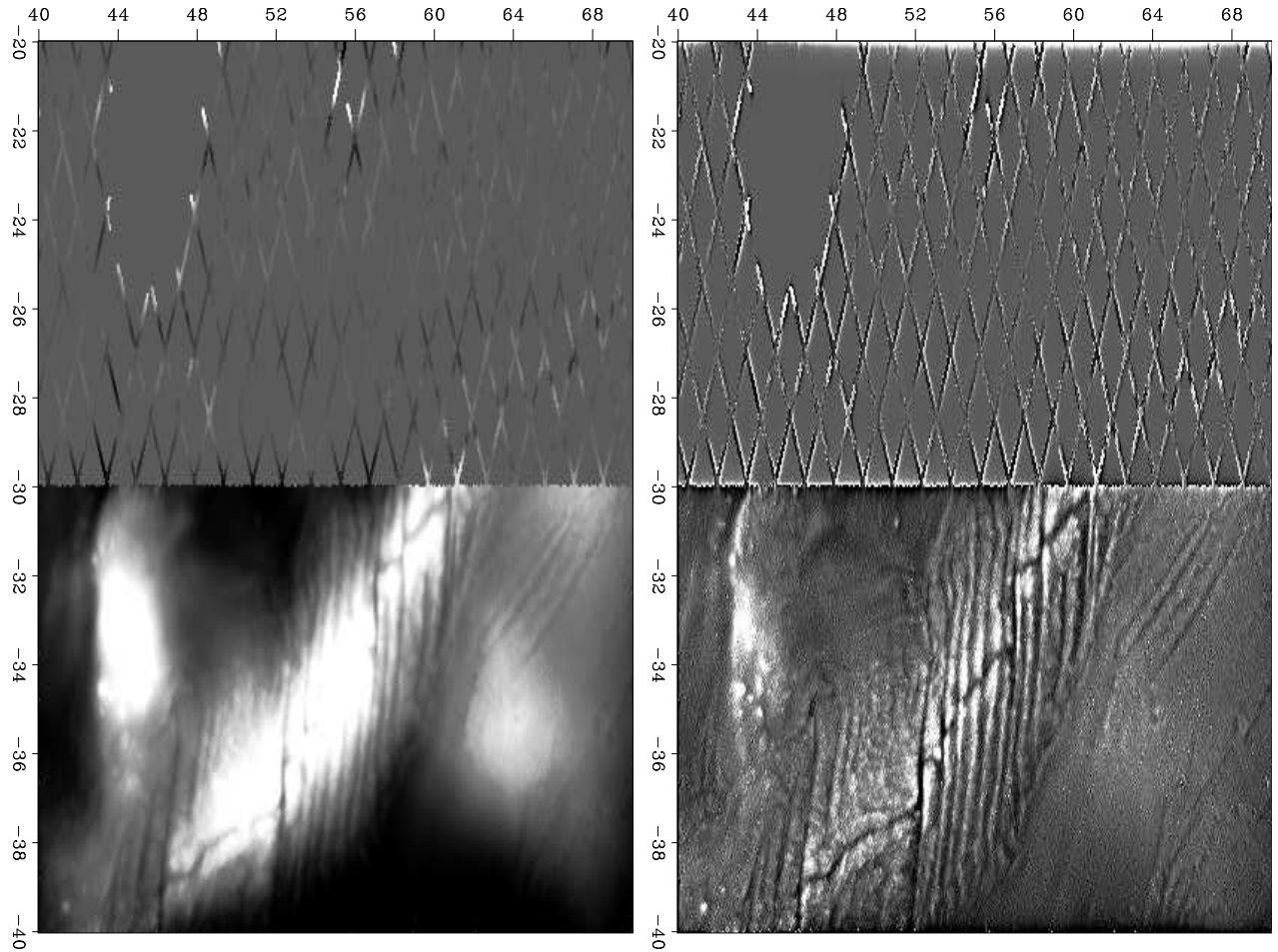


Figure 7.13: All data merged into a track-free image (hooray!) by applying the track derivative, not to the data, but to the residual. Left is $\mathbf{h}$ estimated by $\mathbf{0} \approx \mathbf{W}\frac{d}{dt}(\mathbf{Lh} - \mathbf{d})$. Right is the roughened altitude, $\mathbf{p} = \mathbf{Hh}$. noiz-jesse10 [ER,M]

smooth for nice viewing by variable brightness, but roughening it with $\mathbf{H}$ makes an attractive image showing, in the south, no visible tracks.

The north is another story. We would like the sparse northern tracks to contribute to our viewing pleasure. We would like them to contribute to a northern image of the earth, not to an image of the data acquisition footprint. This begins to happen in Figure 7.14. The process of fitting data by choosing an altitude function $\mathbf{h}$ would normally include some regularization (model styling), such as $\mathbf{0} \approx \nabla \mathbf{h}$. Instead we adopt the usual trick of changing to preconditioning variables, in this case $\mathbf{h} = \mathbf{H}^{-1}\mathbf{p}$. As we iterate with the variable $\mathbf{p}$ we watch the images of $\mathbf{h}$ and $\mathbf{p}$ and quit either when we are tired, or more hopefully, when we are best satisified with the image. This subjective choice is rather like choosing the ϵ that is the balance between data fitting goals and model styling goals. The result in Figure 7.14 is pleasing. We have begun building topography in the north that continues in a consistant way with what is in the south. Unfortunately, this topography does fade out rather quickly as we get off the data acquisition tracks.

If we have reason to suspect that the geological style north of the 30th parallel matches that south of it (the stationarity assumption) we can compute a PEF on the south side and use it for interpolation on the north side. This is done in Figure 7.15.

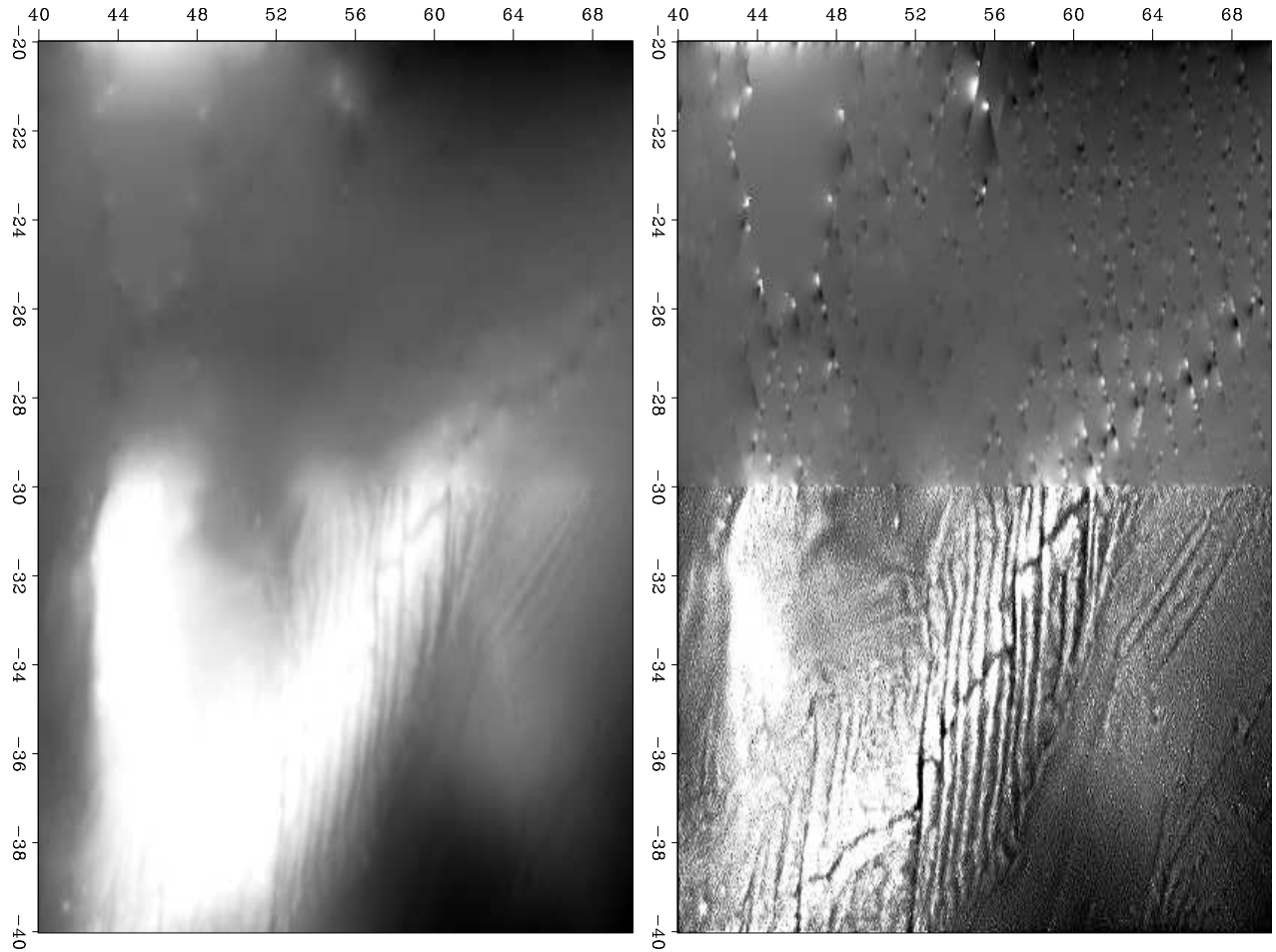


Figure 7.14: Using the track derivative in residual space and helix preconditioning in model space we start building topography in the north. Left is $\mathbf{h} = \mathbf{H}^{-1}\mathbf{p}$ where $\mathbf{p}$ is estimated by $\mathbf{0} \approx \mathbf{W}\frac{d}{dt}(\mathbf{L}\mathbf{H}^{-1}\mathbf{p} - \mathbf{d})$ for only 10 iterations. Right is $\mathbf{p} = \mathbf{H}\mathbf{h}$. noiz-jesse8 [ER,M]

This is about as good as we are going to get. Our fractured ridge continues nicely into the north. Unfortunately, we have imprinted the fractured ridge texture all over the northern space, but that's the price we must pay for relying on the stationarity assumption.

The fitting residuals are shown in Figure 7.16. The physical altitude residuals tend to be rectangles, each the duration of a track. While the satellite is overflying other earth locations the ocean surface is changing its altitude. The fitting residuals (right side) are very fuzzy. They appear to be "white", though with ten thousand points crammed onto a line a couple inches long, we cannot be certain. We could inspect this further. If the residuals turn out to be significantly non-white, we might do better to change $\frac{d}{dt}$ to a PEF along the track.

7.8. ELIMINATING NOISE AND SHIP TRACKS IN GALILEE

The Sea of Galilee data set has enchanted my colleagues and me because the data has comprehensible defects that have frustrated many of our image estimation de-

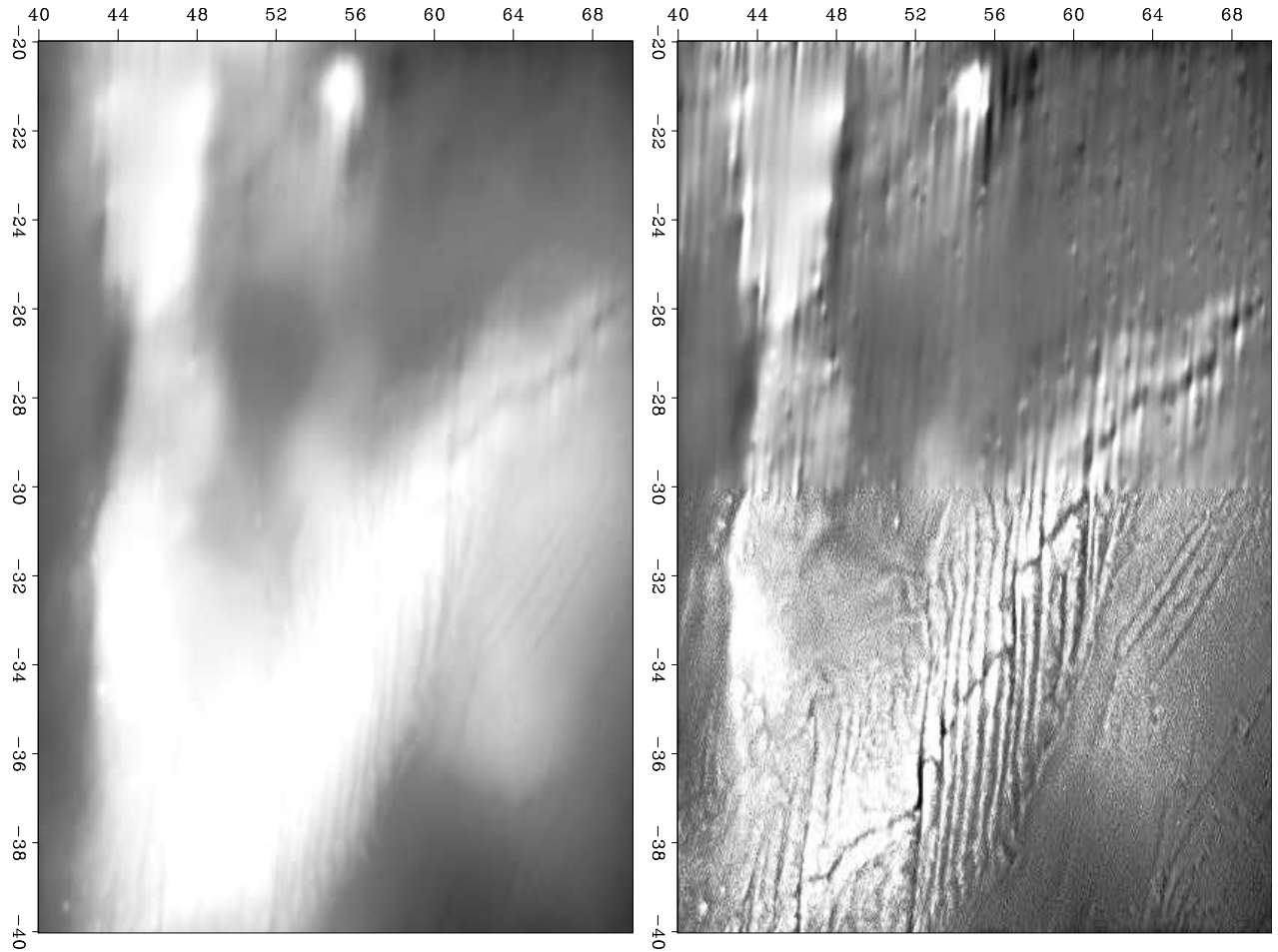


Figure 7.15: Given a PEF $\mathbf{A}$ estimated on the densely defined southern part of the model, $\mathbf{p}$ was estimated by $\mathbf{0} \approx \mathbf{W}\frac{d}{dt}(\mathbf{L}\mathbf{A}^{-1}\mathbf{p} - \mathbf{d})$ for 50 iterations. Left is $\mathbf{h} = \mathbf{A}^{-1}\mathbf{p}$. Right is $\mathbf{p} = \mathbf{H}\mathbf{h}$. noiz-jesse9 [ER,M]

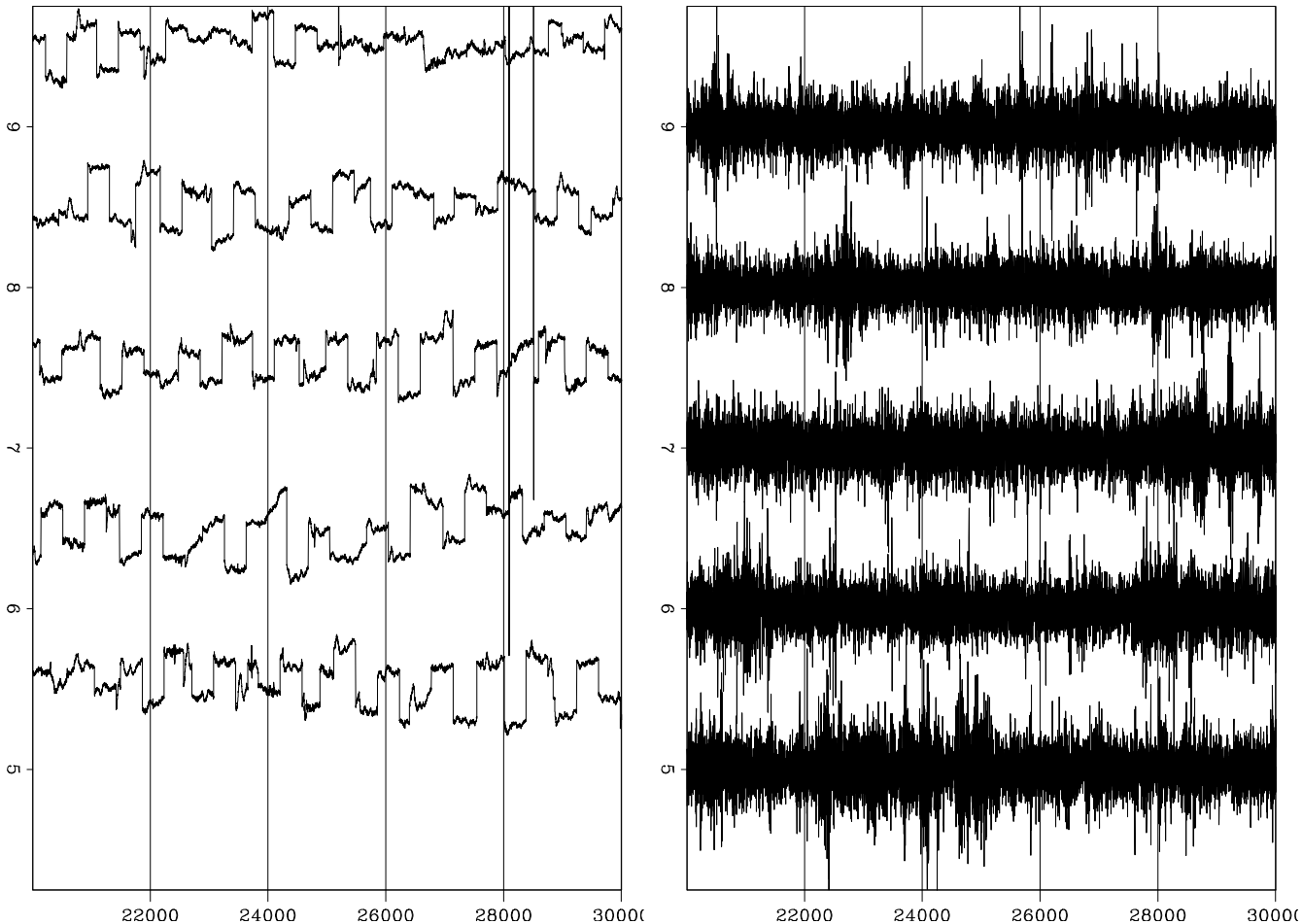


Figure 7.16: The residual at fifty thousand of the half million (537,974) data points in Figure 7.15. Left is physical residual $\mathbf{LA}^{-1}\mathbf{p} - \mathbf{d}$. Right is fitting residual $\mathbf{W}\frac{d}{dt}(\mathbf{LA}^{-1}\mathbf{p} - \mathbf{d})$. noiz-jesse9_res [ER,M]

signs. The best results found so far were prepared for us here by Antoine Guitton based on our 2004 Geophysics paper.

We are given depth-sounding data from the Sea of Galilee. The Sea of Galilee is unique because it is a fresh-water lake below sea-level. It seems to be connected to the Great Rift (pull-apart) valley crossing East Africa. The ultimate goal is to produce a good map of the depth to bottom, and images useful for identifying archaeological, geological, and geophysical details of the water bottom. In particular, we hope to identify some ancient shorelines around the lake and meaningful geological features inside the lake. The ancient shorelines might reveal early settlements of archeological interest or old fishing ports. The pertinence of this data set to our daily geophysical problems is threefold: (1) We often need to interpolate irregular data. (2) The data has noise bursts of various types. (3) The data has systematic error (drift) which tends to leave data-acquisition tracks in the resulting image.

The Galilee data set was introduced in chapter 3 and recently plotted in Figure 7.4. Actually, that figure is a view of 2-D model space. One of the first things I learned (the hard way) is the importance of viewing both the model space and the residuals in data space.

Be sure to plot both model space and data space. You should try to understand the results in both spaces and might like to watch movies of each as the iteration progresses.

The raw data (Figure 7.17), is distributed irregularly across the lake surface. It is 132,044 triples (x_i, y_i, z_i), where x_i ranges over about 12 km, where y_i ranges over about 20 km, and z_i is depth in multiples of 10 cm. (It would have been helpful if a fourth value had been included, the clock-date time t_i, of the measurement.) The ship surveyed a different amount of distance every day of the survey. Figure 7.17 displays the whole survey as one long track. On one traverse across the lake, the depth record is U shaped. A few V shaped tracks result from deep-water vessel turn arounds. All depth values (data points) used for building the final map are shown here. Each point corresponds to one depth measurement inside the lake. For display convenience, the long signal is broken into 23 strips of 5718 depth measurements. We have no way to know that sometimes the ship stops a little while with the data recorder running; sometimes it shuts down overnight or longer; but mostly it progresses at some unknown convenient speed. So the horizontal axis in data space is a measurement number that scales in some undocumented way to

distance along the track.

7.8.1. Attenuation of noise bursts and glitches

Let **h** be an abstract vector containing as components the water depth over a 2-D spatial mesh. Let **d** be an abstract vector whose successive components are depths along the vessel tracks. One way to grid irregular data is to minimize the length of the residual vector $\mathbf{r}_d(\mathbf{h})$:

$$\mathbf{0} \quad \approx \quad \mathbf{r}_d \quad = \quad \mathbf{Bh} - \mathbf{d} \tag{7.20}$$

where **B** is a 2-D linear interpolation (or binning) operator and $\mathbf{r}_d$ is the data residual. Where tracks cross or where multiple data values end up in the same bin, the fitting goal (7.20) takes an average. Figure 7.4 is a display of simple binning of the raw data. (Some data points are outside the lake. These must represent navigation errors.)

Some model-space bins will be empty. For them we need an additional "model

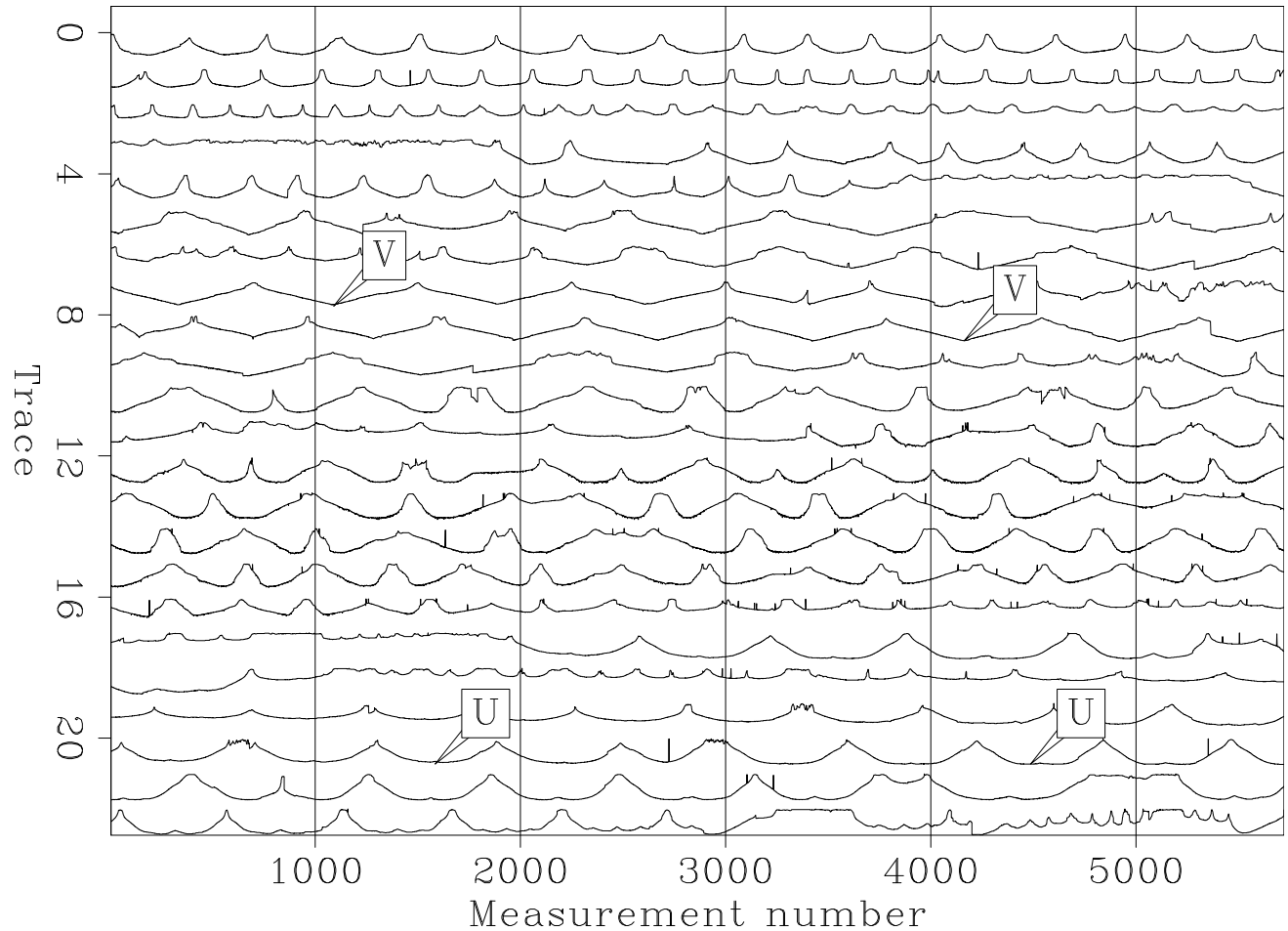


Figure 7.17: The complete Galilee data space. noiz-antoine1 [ER]

styling" goal, i.e. regularization. For simplicity we might minimize the gradient.

$$\begin{array}{rcccl} \mathbf{0} & \approx & \mathbf{r}_d & = & \mathbf{Bh} - \mathbf{d} \\ \mathbf{0} & \approx & \mathbf{r}_h & = & \epsilon \nabla \mathbf{h} \end{array} \tag{7.21}$$

where $\nabla = \left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y} \right)$ and $\mathbf{r}_h$ is the model space residual. Choosing a large scaling factor ϵ will tend to smooth our entire image, not just the areas of empty bins. We would like ϵ to be any number small enough that its main effect is to smooth areas of empty bins. When we get into this further, though, we'll see that because of noise some smoothing across the nonempty bins is desireable too.

7.8.2. Preconditioning for accelerated convergence

As usual we precondition by changing variables so that the regularization operator becomes an identity matrix. The gradient ∇ in equation (7.21) has no inverse, but its spectrum $-\nabla'\nabla$, can be factored ($-\nabla'\nabla = \mathbf{H}'\mathbf{H}$) into triangular parts $\mathbf{H}$ and $\mathbf{H}'$ where $\mathbf{H}$ is the helix derivative. This $\mathbf{H}$ is invertible by deconvolution. The quadratic form $\mathbf{h}'\nabla'\nabla\mathbf{h} = \mathbf{h}'\mathbf{H}'\mathbf{H}\mathbf{h}$ suggests the new preconditioning variable $\mathbf{p} = \mathbf{H}\mathbf{h}$. The

fitting goals in equation (7.21) thus become

$$\begin{array}{rcll} \mathbf{0} & \approx & \mathbf{r}_d & = \quad \mathbf{B}\mathbf{H}^{-\mathbf{1}}\mathbf{p} - \mathbf{d} \\ \mathbf{0} & \approx & \mathbf{r}_p & = \quad \epsilon\mathbf{p} \end{array} \tag{7.22}$$

with $\mathbf{r}_p$ the residual for the new variable $\mathbf{p}$. Experience shows that an iterative solution for $\mathbf{p}$ converges much more rapidly than an iterative solution for $\mathbf{h}$, thus showing that $\mathbf{H}$ is a good choice for preconditioning. We could view the estimated final map $\mathbf{h} = \mathbf{H}^{-\mathbf{1}}\mathbf{p}$, however in practice, because the depth function is so smooth, we usually prefer to view the roughened depth $\mathbf{p}$.

There is no simple way of knowing beforehand the best value of ϵ. Practitioners like to see solutions for various values of ϵ. Practical exploratory data analysis is pragmatic. Without a simple, clear theoretical basis, analysts generally begin from $\mathbf{p} = \mathbf{0}$ and then abandon the fitting goal $\mathbf{0} \approx \mathbf{r}_p = \epsilon\mathbf{p}$. Effectively, they take $\epsilon = 0$. Then they examine the solution as a function of iteration, imagining that the solution at larger iterations corresponds to smaller ϵ and that the solution at smaller iterations corresponds to larger ϵ. In all our explorations, we follow this approach and omit the regularization in the estimation of the depth maps. Having achieved the general results we want, we should include the parameter ϵ and adjust it until we see a

pleasing result at an "infinite" number of iterations. We should but usually we do not.

7.8.3. ℓ^1 norm

Spikes and erratic noise glitches can be suppressed with an approximate ℓ^1 norm. One main problem with the Galilee data is the presence of outliers in the middle of the lake and at the track ends. We could attenuate these spikes by editing or applying running median filters. However, the former involves human labor while the latter might compromise small details by smoothing and flattening the signal. Here we formulate the estimation to eliminate the drastic effect of the noise spikes. We introduce a weighting operator that deemphasizes high residuals as follows:

$$\begin{aligned} \mathbf{0} &\approx \mathbf{r}_d &&= \mathbf{W}(\mathbf{B}\mathbf{H}^{-\mathbf{1}}\mathbf{p} - \mathbf{d}) \\ \mathbf{0} &\approx \mathbf{r}_p &&= \epsilon\mathbf{p} \end{aligned} \tag{7.23}$$

with a diagonal matrix $\mathbf{W}$:

$$\mathbf{W} = \mathbf{diag}\left(\frac{1}{(1+r_i^2/\bar{r}^2)^{1/4}}\right) \tag{7.24}$$

where r_i is the residual for one component of $\mathbf{r}_d$ and $\bar{r}$ is a prechosen constant. This weighting operator ranges from ℓ^2 to ℓ^1, depending on the constant $\bar{r}$. We take $\bar{r} = 10$ cm because the data was given to us as integer multiples of 10 cm. (A somewhat larger value might be more appropriate).

Figure 7.18 displays $\mathbf{p}$ estimated in a least-squares sense on the left and in a ℓ^1 sense on the right (equation (7.23) with a small $\bar{r}$). Most of the glitches are no longer visible. One obvious glitch remains near $(x, y) = (205, 238)$. Evidently a north-south track has a long sequence of biased measurements that our ℓ^1 cannot overcome. Some ancient shorelines in the western and southern parts of the Sea of Galilee are now easier to identify (shown as AS). We also start to see a valley in the middle of the lake (shown as R). Data outside the lake (navigation errors) have been mostly removed. Data acquisition tracks (mostly north-south lines and east-west lines, one of which is marked with a T) are even more visible after the suppression of the outliers.

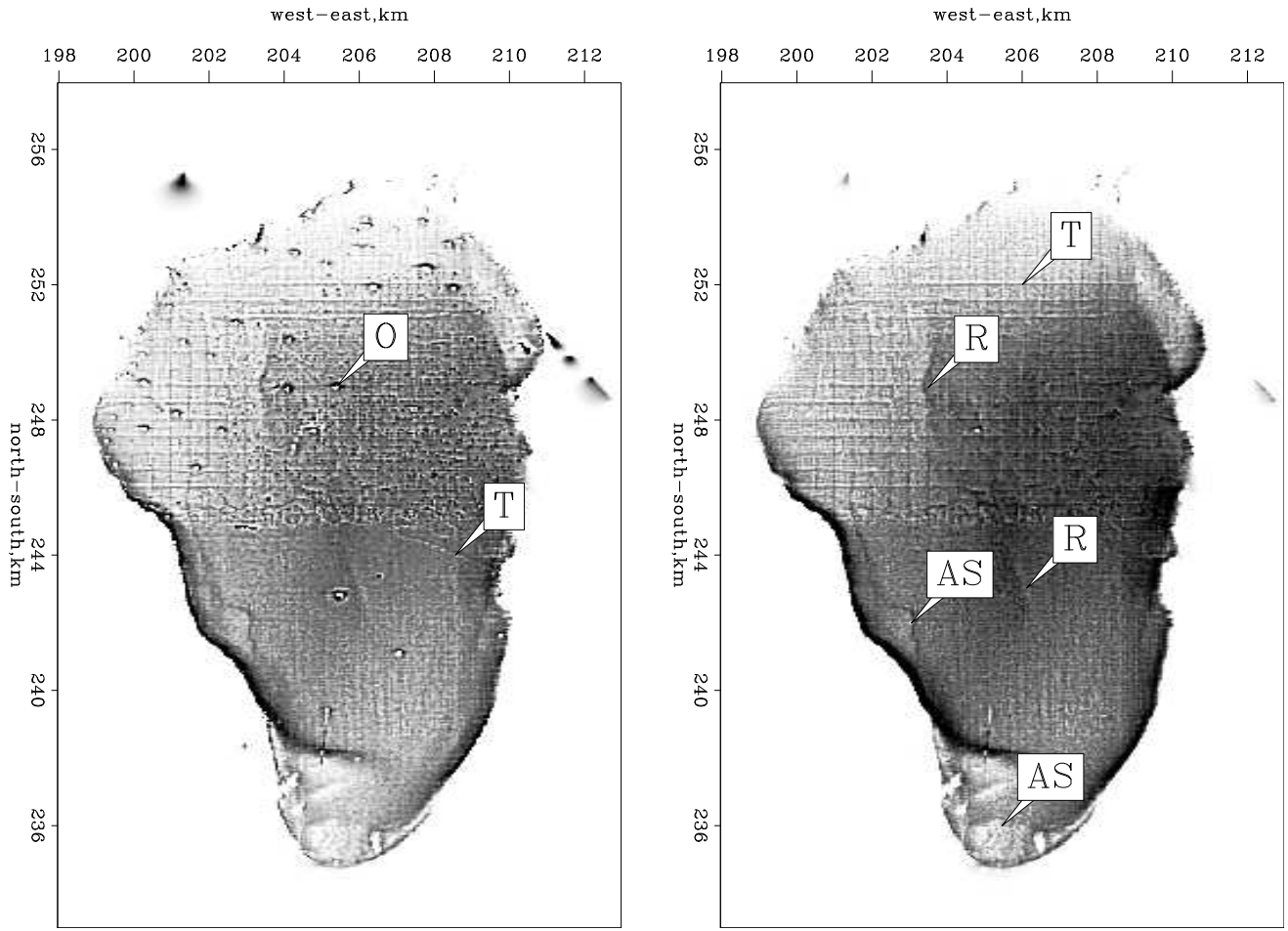


Figure 7.18: Estimated **p** in a least-squares sense (left) and in an ℓ^1 sense (right). Pleasingly, isolated spikes are attenuated. Some interesting features are shown by the arrows: AS points to few ancient shores, O points to some outliers, T points to few tracks, and R points to a curious feature. noiz-antoine2 [ER,M]

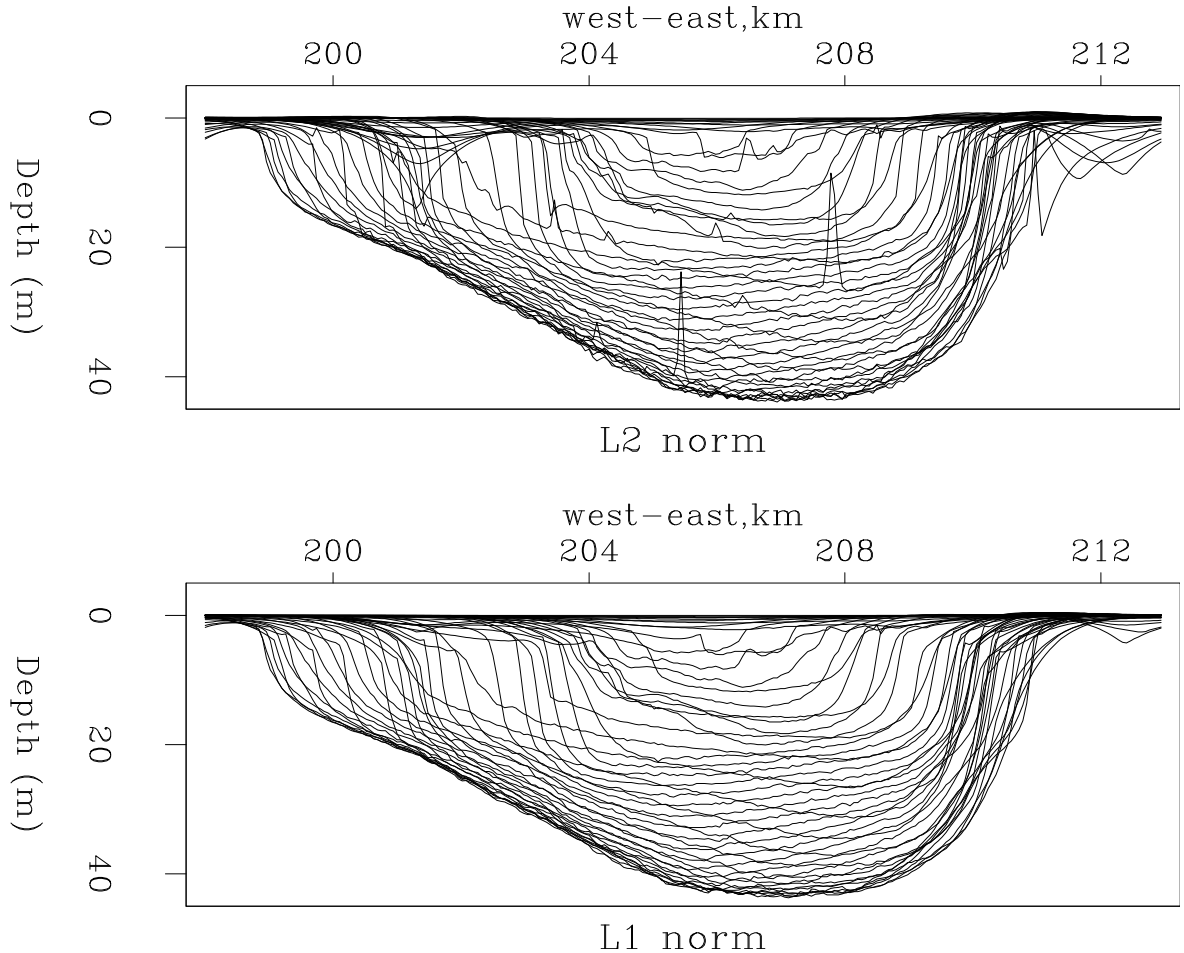


Figure 7.19: East-west cross sections of the lake bottom ($\mathbf{h} = \mathbf{H}^{-1}\mathbf{p}$). Top with the ℓ^2 solution. Bottom with the ℓ^1 approximating procedure. noiz-antoine3 [ER,M]

Figure 7.19 shows the bottom of the Sea of Galilee ($\mathbf{h} = \mathbf{H}^{-1}\mathbf{p}$) with ℓ^2 (top) fitting and ℓ^1 (bottom) fitting. Each line represents one east-west transect, transects at half-kilometer intervals on the north-south axis. The ℓ^1 result is a nice improvement over the ℓ^2 maps. The glitches inside and outside the lake have mostly disappeared. Also, the ℓ^1 norm gives positive depths everywhere. Although not visible everywhere in all the figures, topography is produced outside the lake. Indeed, the effect of regularization is to produce synthetic topography, a natural continuation of the lake floor surface.

We are now halfway to a noise-free image. Figure 7.18 shows that vessel tracks overwhelm possible fine scale details. Next we investigate a strategy based on the idea that the inconsistency between tracks comes mainly from different human and seasonal conditions during the data acquisition. Since we have no records of the weather and the time of the year the data were acquired we presume that the depth differences between different acquisition tracks must be small and relatively smooth along the super track.

7.8.4. Abandoned strategy for attenuating tracks

An earlier strategy to remove the ship tracks is to filter the residual as follows:

$$\begin{array}{rcll} \mathbf{0} & \approx & \mathbf{r}_d & = \mathbf{W}\frac{d}{ds}(\mathbf{B}\mathbf{H}^{-\mathbf{1}}\mathbf{p}-\mathbf{d}), \\ \mathbf{0} & \approx & \mathbf{r}_p & = \epsilon\mathbf{p}, \end{array} \tag{7.25}$$

where $\frac{d}{ds}$ is the derivative along the track. The derivative removes the drift from the field data (and the modeled data). An unfortunate consequence of the track derivative is that it creates more glitches and spiky noise at the track ends and at the bad data points. Several students struggled with this idea without good results.

One explanation (of unknown validity) given for the poor results is that perhaps the numerical conditioning of the algebraic problem is worsened by the operators $\mathbf{W}$, $\frac{d}{ds}$, $\mathbf{B}$, and $\mathbf{H}^{-\mathbf{1}}$, drastically slowing the convergence. Another explanation is that the operator $\frac{d}{ds}$ is too simple. Perhaps we should have a five or ten point low-cut filter—or maybe a PEF. A PEF could be estimated from the residual itself. Unfortunately, such a longer filter would smear the bad effect of noise glitches onto more residuals, effectively spoiling more measurements.

We concluded that the data is bad only in a very low frequency sense. Perhaps the lake is evaporating, or it is raining, or the load in the boat has been changed or

shifted. It's a fact that any very low-frequency reject filter is necessarily a long filter, and that means that it must catch many noise spikes. Thus we should not attempt to filter out the drift from the residual. Instead we should model the drift.

In the presence of both noise bursts and noise with a sensible spectrum (systematic noise), the systematic noise should be modeled while the noise bursts should be handled with ℓ^1.

7.8.5. Modeling data acquisition drift

To model data drift we imagine a vector $\mathbf{q}$ of random numbers that will be passed thru a low-pass filter (like a leaky integrator) $\mathbf{L}$. The modeled data drift is $\mathbf{Lq}$. We will solve for $\mathbf{q}$. A price we pay is an increase of the number of unknowns. Augmenting earlier fitting goals (7.23) we have:

$$\begin{array}{lllll} \mathbf{0} & \approx & \mathbf{r}_d & = & \mathbf{W}(\mathbf{B}\mathbf{H}^{-\mathbf{1}}\mathbf{p} + \lambda\mathbf{L}\mathbf{q} - \mathbf{d}), \\ \mathbf{0} & \approx & \mathbf{r}_p & = & \epsilon_1\mathbf{p}, \\ \mathbf{0} & \approx & \mathbf{r}_q & = & \epsilon_2\mathbf{q}, \end{array} \tag{7.26}$$

where $\mathbf{h} = \mathbf{H}^{-1}\mathbf{p}$ estimates the interpolated map of the lake, and where $\mathbf{L}$ is a drift modeling operator (leaky integration), $\mathbf{q}$ is an additional variable to be estimated, and λ is a balancing constant to be discussed. We then minimize the misfit function,

$$g_2(\mathbf{p},\mathbf{q}) = \|\mathbf{r}_d\|^2 + \epsilon_1^2\|\mathbf{r}_p\|^2 + \epsilon_2^2\|\mathbf{r}_q\|^2, \tag{7.27}$$

Now the data $\mathbf{d}$ is being modeled in two ways by two parts which add, a geological map part $\mathbf{BH}^{-1}\mathbf{p}$ and a recording system drift part $\lambda\mathbf{Lq}$. Clearly, by adjusting the balance of ϵ_1 to ϵ_2 we are forcing the data to go one way or the other. There is nothing in the data itself that says which part of the theory should claim it.

It is a customary matter of practice to forget the two ϵs and play with the λ. If we kept the two ϵs, the choice of λ would be irrelevant to the final result. Since we are going to truncate the iteration, choice of λ matters. It chooses how much data energy goes into the equipment drift function and how much into topography. Antoine ended out with with $\lambda = 0.08$.

There is another parameter to adjust. The parameter ρ controlling the decay of the leaky integration. Antoine found that value $\rho = 0.99$ was a suitable compromise. Taking ρ smaller allows the track drift to vary too rapidly thus falsifying data in a way that falsifies topography. Taking ρ closer to unity does not allow adequately

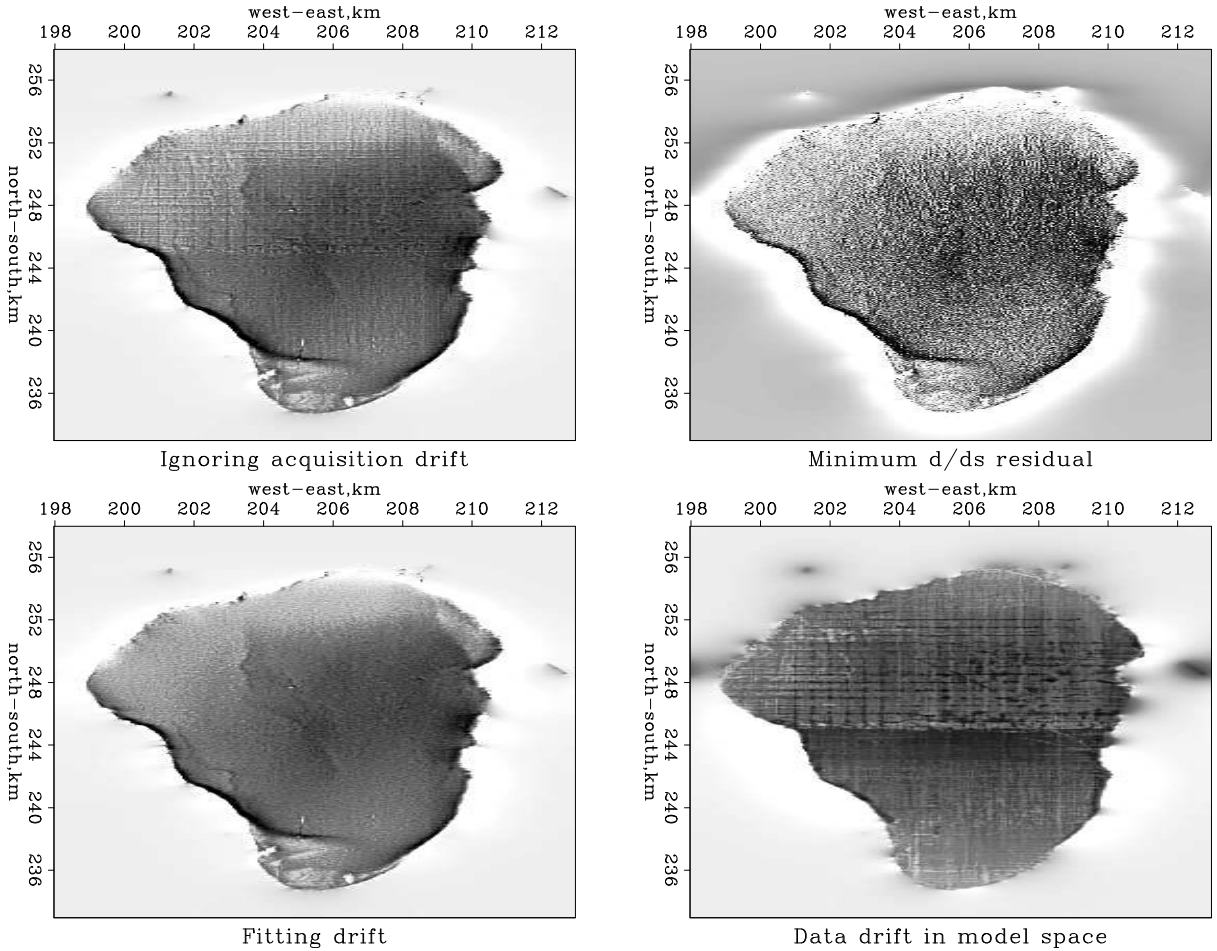


Figure 7.20: Top left: Estimated **p** without attenuation of the tracks, i.e., regression (7.23). Top right: Estimated **p** with the derivative along the tracks, i.e., regression (7.25). Bottom left: Estimated **p** without tracks, i.e., regression (7.26). Bottom right: recorder drift in model space $\mathbf{B}'\mathbf{Lq}$. noiz-antoine4 [ER,M]

rapid variation of the data acquistion system thereby pushing acquisition tracks into the topography.

Figure 7.20 (bottom-left corner) shows the estimated roughened image $\mathbf{p}$ with $\lambda\mathbf{L}$ data-drift modeling and (top-left corner) $\mathbf{p}$ without it. Data-drift modeling (bottom-left) yields an image that is essentially track-free without loss of detail. Top right shows the poor result of applying the derivative $\frac{d}{ds}$ along the tracks. Tracks are removed but the topography is unclear.

The bottom-right part of Figure 7.20 provides important diagnostic information. The estimated instrumentation drift $\mathbf{Lq}$ has been transformed to model space $\mathbf{B'Lq}$. We do not like to see hints of geology in this space but we do. Adjusting λ or ρ we can get rid of the geology here, but then survey tracks will appear in the lake image. The issue of decomposing data into signal and noise parts is dealt with further in chapter 9.

Figures 7.21 and 7.22 show selected segments of data space. Examining here the discrepancy between observed data and modeled data offers us an opportunity to get new ideas. The top plot is the input data $\mathbf{d}$. Next is the estimated noise-free data $\mathbf{BH^{-1}p}$. Then the estimated secular variations $\lambda\mathbf{Lq}$. Finally residual $\mathbf{BH^{-1}p} + \lambda\mathbf{Lq} - \mathbf{d}$ after a suitable number of iterations. The modeled data in both Figures

7.21b and 7.22b show no remaining spikes.

The estimated instrument drift is reasonable, mostly under a meter for measurments with a nominal precision of 10 cm. There are some obvious problems though. It is not a serious problem that the drift signal is always positive. Applying the track derivative means that zero frequency is in the null space. An arbitrary constant may be moved from water depth to track calibration. More seriously, the track calibration fluctuates more rapidly than we might imagine. Worse still, Figure 7.22c shows the instrument drift correlates with water depth(!). This suggests we should have a slower drift function (bigger ρ or weaker λ), but Antoine assures me that this would push data acquisition tracks into the lake image. If the data set had included the date-time of each measurement we would have been better able to model drift. Instead of allowing a certain small change of drift with each measurement, we could have allowed a small change in proportion to the time since the previous measurement.

An interesting feature of the data residual in Figure 7.22d is that it has more variance in deep water than in shallow. Perhaps the depth sounder has insufficient power for deeper water or for the softer sediments found in deeper water. On the other hand, this enhanced deep water variance is not seen in Figure 7.21d which is

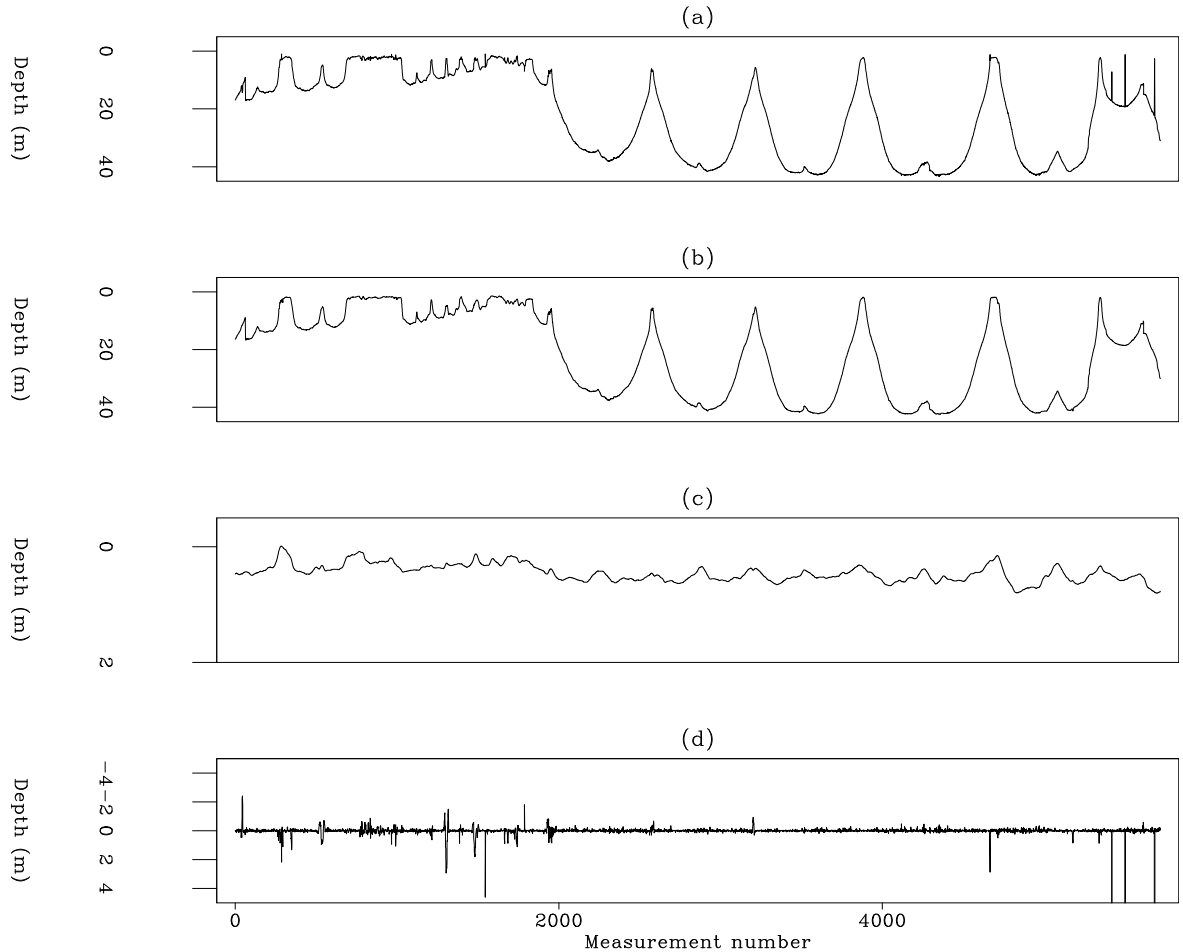


Figure 7.21: (a) Track 17 (input data) in Figure 7.17. (b) The estimated noise-free data $\mathbf{BH}^{-1}\mathbf{p}$. (c) Estimated drift $\mathbf{Lq}$. (d) Data residual. noiz-antoine5 [ER,M]

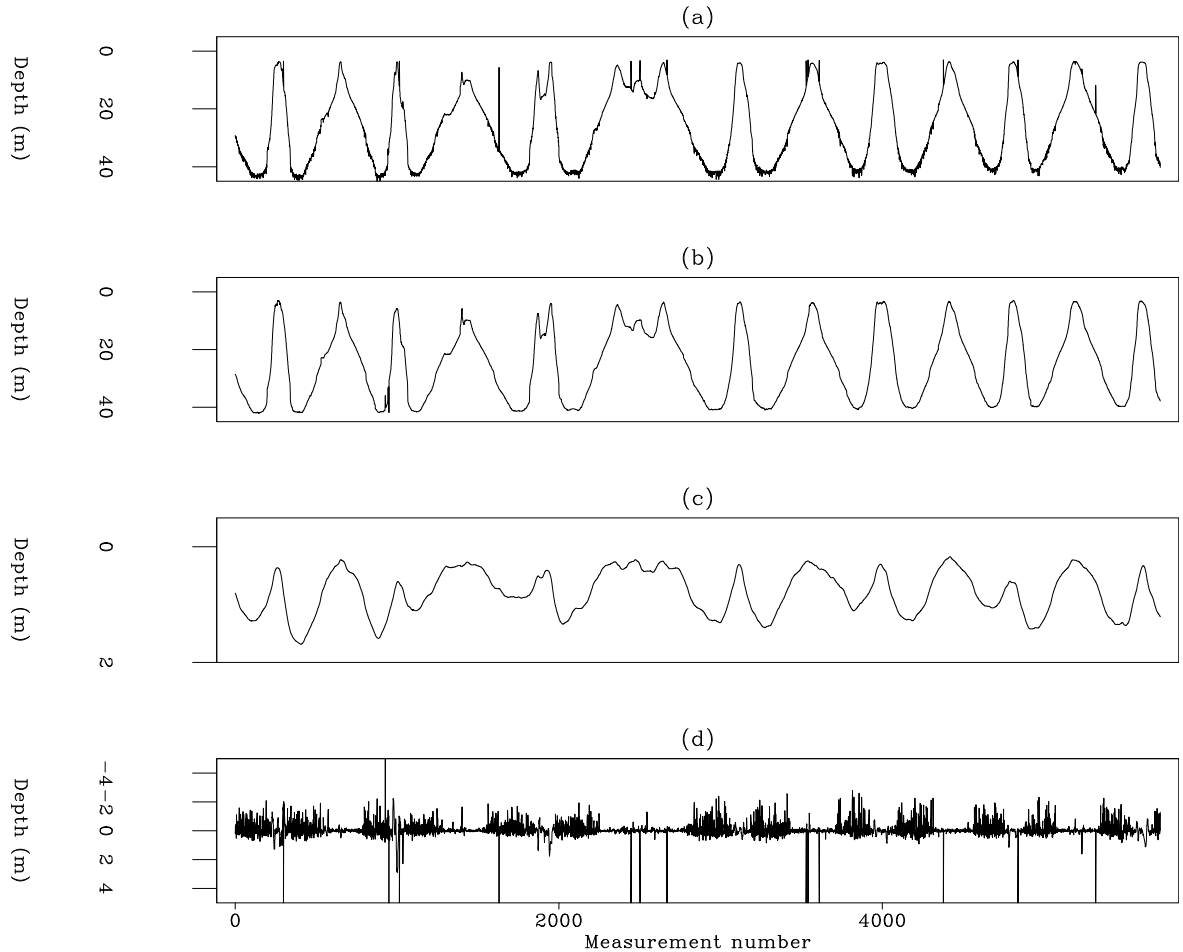


Figure 7.22: (a) Track 14 (input data) in Figure 7.17. (b) Modeled data, $\mathbf{BH^{-1}p}$. (c) Estimated drift. (d) Data-space residual. noiz-antoine6 [ER]

puzzling. Perhaps the sea was rough for one day of recording but not another.

7.8.6. Regridding

We often have an image $\mathbf{h}_0$ on a coarse mesh that we would like to use on a refined mesh. This regridding chore reoccurs on many occasions so I present reusable code. When a continuum is being mapped to a mesh, it is best to allocate to each mesh point an equal area on the continuum. Thus we take an equal interval between each point, and a half an interval beyond the end points. Given `n` points, there are `n-1` intervals between them, so we have

```
min = o - d/2
max = o + d/2 + (n-1)*d
```

which may be back solved to

```
d = (max-min)/n
o = (min*(n-.5) + max/2)/n
```

which is a memorable result for `d` and a less memorable one for `o`. With these not-quite-trivial results, we can invoke the linear interpolation operator `lint2`. It is

designed for data points at irregular locations, but we can use it for regular locations too. Operator `refine2` defines pseudoirregular coordinates for the bin centers on the fine mesh and then invokes `lint2` to carry data values from the coarse regular mesh to the pseudoirregular finer mesh. Upon exiting from `refine2`, the data space (normally irregular) is a model space (always regular) on the finer mesh. refine2

Finally, here is the 2-D linear interpolation operator `lint2`, which is a trivial extension of the 1-D version `lint1` /prog:lint1. lint2

```
module refine2 { # Refine mesh.
# Input mm(m1,m2) is coarse.  Output dd(n1,n2) linear interpolated.
#
use lint2
real, dimension( :, :), pointer, private :: xy
#% _init( co1,cd1,co2,cd2, m1,m2, fo1,fd1,fo2,fd2, n1,n2)
  integer, intent( in)  :: m1,m2, n1,n2
  real,    intent( in)  :: co1,cd1,co2,cd2                # coarse grid
  real,    intent( out) :: fo1,fd1,fo2,fd2                # fine grid
  integer               :: i1,i2, id
  real                  :: xmin,xmax, ymin,ymax, x,y
  allocate (xy( n1*n2, 2))
  xmin = co1-cd1/2;   xmax = co1 +cd1*(m1-1) +cd1/2       # Great formula!
  ymin = co2-cd2/2;   ymax = co2 +cd2*(m2-1) +cd2/2
  fd1= (xmax-xmin)/n1;  fo1= (xmin*(n1-.5) + xmax/2)/n1   # Great formula!
  fd2= (ymax-ymin)/n2;  fo2= (ymin*(n2-.5) + ymax/2)/n2
  do i2=1,n2 {  y = fo2 + fd2*(i2-1)
  do i1=1,n1 {  x = fo1 + fd1*(i1-1)
        id = i1+n1*(i2-1)
        xy( id, :) = (/ x, y /)
        }}
  call lint2_init( m1,m2, co1,cd1, co2,cd2, xy)
#% _lop (mm, dd)
  integer stat1
  stat1 = lint2_lop( adj, .true., mm, dd)
#% _close
deallocate (xy)
}
```

Back

```
module lint2 {                                          # (Bi)Linear interpola-
tion in 2-D
integer                         :: m1,m2
real                            :: o1,d1, o2,d2
real, dimension (:,:), pointer :: xy
#%  _init (     m1,m2, o1,d1, o2,d2, xy)
#%  _lop  ( mm (m1,m2), dd (:))
integer i, ix,iy, id
real    f, fx,gx, fy,gy
do id= 1, size(dd) {
   f = (xy(id,1)-o1)/d1; i=f; ix= 1+i; if( 1>ix .or. ix>=m1) cycle; fx=f-
i; gx= 1.-fx
   f = (xy(id,2)-o2)/d2; i=f; iy= 1+i; if( 1>iy .or. iy>=m2) cycle; fy=f-
i; gy= 1.-fy
                if( adj) {
                        mm(ix  ,iy  ) += gx * gy * dd(id)
                        mm(ix+1,iy  ) += fx * gy * dd(id)
                        mm(ix  ,iy+1) += gx * fy * dd(id)
                        mm(ix+1,iy+1) += fx * fy * dd(id)
                        }
                else
                        dd(id) = dd(id) + gx * gy * mm(ix  ,iy  ) +
                                          fx * gy * mm(ix+1,iy  ) +
                                          gx * fy * mm(ix  ,iy+1) +
                                          fx * fy * mm(ix+1,iy+1)
   }
}
```

Back

Chapter 8

Spatial aliasing and scale invariance

Landforms are not especially predictable. Therefore, crude PEF approximations are often satisfactory. Wavefields are another matter. Consider the “shape” of the

acoustic wavefronts at this moment in the room you are in. The acoustic wavefield has statistical order in many senses. If the 3-D volume is filled with waves emitted from a few point sources, then (with some simplifications) what could be a volume of information is actually a few 1-D signals. When we work with wavefronts we can hope for more dramatic, even astounding, results from estimating properly.

The plane-wave model links an axis that is not aliased (time) with axes (space) that often are.

We often characterize data from any region of (t, x)-space as "good" or "noisy" when we really mean it contains "few" or "many" plane-wave events in that region. Where regions are noisy, there is no escaping the simple form of the Nyquist limitation. Where regions are good we may escape it. Real data typically contains both kinds of regions. Undersampled data with a broad distribution of plane waves is nearly hopeless. Undersampled data with a sparse distribution of plane waves offer us the opportunity to resample without aliasing. Consider data containing a spherical wave. The angular bandwidth in a plane-wave decomposition appears huge *until we restrict attention to a small region* of the data. (Actually a spherical wave contains very little information compared to an arbitrary wave field.) It can be very

helpful in reducing the local angular bandwidth if we can deal effectively with tiny pieces of data. If we can deal with tiny pieces of data, then we can adapt to rapid spatial and temporal variations. This chapter shows such tiny windows of data.

8.1. INTERPOLATION BEYOND ALIASING

A traditional method of data interpolation on a regular mesh is a four-step procedure: (1) Set zero values at the points to be interpolated; (2) Fourier transform; (3) Set to zero the high frequencies; and (4) Inverse transform. This is a fine method and is suitable for many applications in both one dimension and higher dimensions. However, this method fails to take advantage of our prior knowledge that seismic data has abundant fragments of plane waves that link an axis that is not aliased (time) to axes that often are (space).

8.1.1. Interlacing a filter

The filter below can be designed despite alternate missing traces. This filter destroys plane waves. If the plane wave should happen to pass halfway between the “d”

and the "e", those two points could interpolate the halfway point, at least for well-sampled temporal frequencies, and the time axis should always be well sampled. For example, $d = e = -.5$ would almost destroy the plane wave and it is an aliased planewave for its higher frequencies.

$$\begin{array}{ccccccccc} a & \cdot & b & \cdot & c & \cdot & d & \cdot & e \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot \end{array} \tag{8.1}$$

We could use module `pef` /prog:pef to find the filter (8.1), if we set up the lag table `lag` appropriately. Then we could throw away alternate zeroed rows and columns (rescale the lag) to get the filter

$$\begin{array}{ccccc} a & b & c & d & e \\ \cdot & \cdot & 1 & \cdot & \cdot \end{array} \tag{8.2}$$

which could be used with subroutine `mis1()` /prog:mis1, to find the interleaved data because both the filters (8.1) and (8.2) have the same dip characteristics.

Figure 8.1 shows three plane waves recorded on five channels and the interpolated data. Both the original data and the interpolated data can be described as

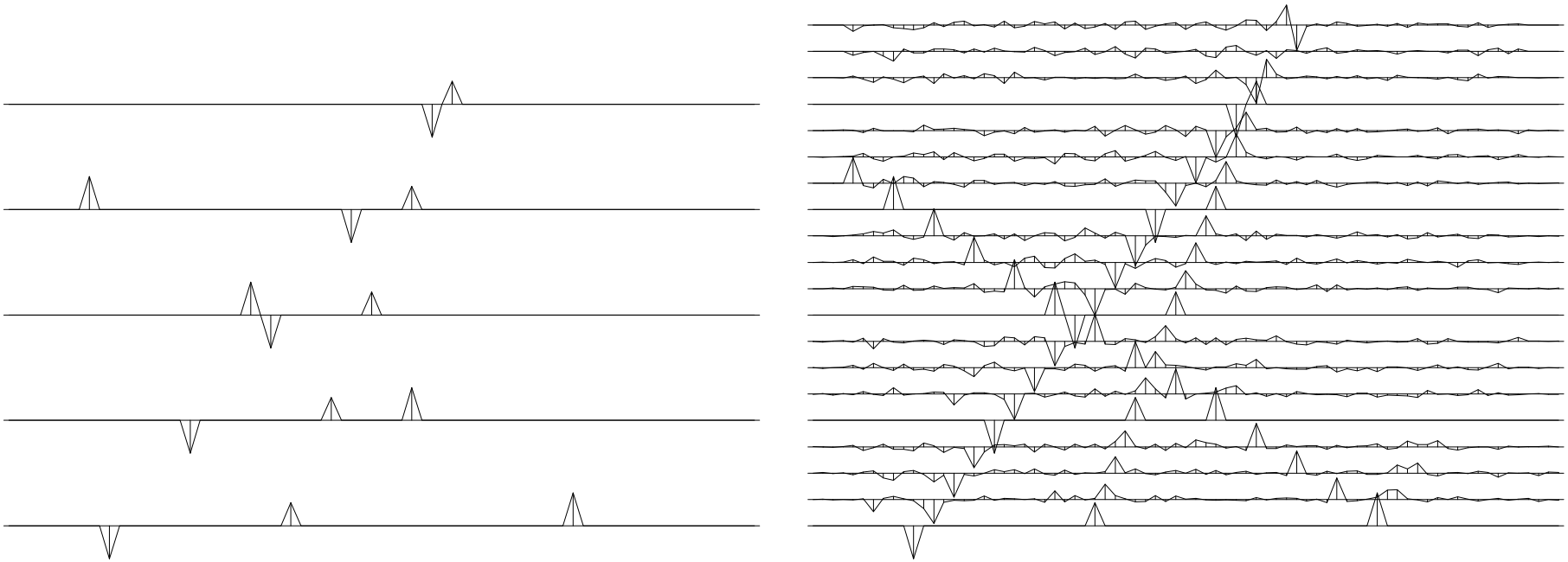


Figure 8.1: Left is five signals, each showing three arrivals. With the data shown on the left (and no more), the signals have been interpolated. Three new traces appear between each given trace, as shown on the right. lal-lace390 [ER]

"beyond aliasing," because on the input data the signal shifts exceed the signal duration. The calculation requires only a few seconds of a two-stage least-squares method, in which the first stage estimates a PEF (inverse spectrum) of the known data, and the second uses the PEF to estimate the missing traces. Figure 8.1 comes from PVI which introduces the clever method described above. We will review how that was done and examine the F90 codes that generalize it to N-dimensions. Then we'll go on to more general methods that allow missing data in any location. Before the methods of this section are applied to field data for migration, data must be broken into many overlapping tiles of size about like those shown here and the results from each tile pieced together. That is described later in chapter 9.

A PEF is like a differential equation. The more plane-wave solutions you expect, the more lags you need on the data. Returning to Figure 8.1, the filter must cover four traces (or more) to enable it to predict three plane waves. In this case, `na=(9,4)`. As usual, the spike on the 2-D PEF is at `center=(5,1)`. We see the filter is expanded by a factor of `jump=4`. The data size is `nd=(75,5)` and `gap=0`. Before looking at the code `lace` /prog:lace for estimating the PEF, it might be helpful to recall the basic utilities `line2cart` and `cart2line` /prog:cartesian for conversion between a multidimensional space and the helix filter lag. We need to sweep across

```
module lace {                                   # find PEF on interlaced data
  use createhelixmod
  use bound
  use pef
  use cartesian
contains
  function lace_pef( dd, jump, nd, center, gap, na)  result( aa) {
    type( filter)                         :: aa
    integer,               intent( in) :: jump
    integer, dimension(:), intent( in) :: nd, center, gap, na
    real,    dimension(:), pointer     :: dd          # input data
    integer, dimension(:), pointer     :: savelags    # holding place
    integer, dimension( size( nd))     :: ii
    integer                            :: ih, nh, lag0, lag
    aa = createhelix( nd, center, gap, na);  nh = size( aa%lag)
    savelags => aa%lag;   allocate( aa%lag( nh))      # prepare interlaced helix
    call cart2line(  na, center, lag0)
    do ih = 1, nh {                                   # Sweep thru the filter.
       call line2cart( na, ih+lag0, ii)
       ii = ii - center;              ii(1) = ii(1)*jump  # Interlace on 1-axis.
       call cart2line( nd,  ii+1, lag)
       aa%lag( ih) = lag - 1
       }
    call boundn( nd, nd, (/ na(1)*jump, na(2:) /), aa)  # Define  aa.mis
    call find_pef( dd, aa, nh*2)                        # Estimate aa coefs
    deallocate( aa%lag);   aa%lag => savelags           # Restore filter lags
  }
}
```

Back

the whole filter and "stretch" its lags on the 1-axis. We do not need to stretch its lags on the 2-axis because the data has not yet been interlaced by zero traces. lace The line `ii(1)=ii(1)*jump` means we interlace the 1-axis but not the 2-axis because the data has not yet been interlaced with zero traces. For a 3-D filter `aa(na1,na2,na3)`, the somewhat obtuse expression `(/na(1)*jump, na(2:)/)` is a three component vector containing `(na1*jump, na2, na3)`.

After the PEF has been found, we can get missing data in the usual way with with module `mis2` /prog:mis2.

8.2. MULTISCALE, SELF-SIMILAR FITTING

Large objects often resemble small objects. To express this idea we use *axis scaling* and we apply it to the basic theory of prediction-error filter (PEF) fitting and missing-data estimation.

Equations (8.3) and (8.4) compute the same thing by two different methods, $\mathbf{r} = \mathbf{Ya}$ and $\mathbf{r} = \mathbf{Ay}$. When it is viewed as fitting goals minimizing $||\mathbf{r}||$ and used along with suitable constraints, (8.3) leads to finding filters and spectra, while (8.4)

leads to finding missing data.

$$\left[\begin{array}{c} r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \\ \hline r_6 \\ r_7 \\ r_8 \\ r_9 \end{array}\right] = \left[\begin{array}{cc} y_2 & y_1 \\ y_3 & y_2 \\ y_4 & y_3 \\ y_5 & y_4 \\ y_6 & y_5 \\ \hline y_3 & y_1 \\ y_4 & y_2 \\ y_5 & y_3 \\ y_6 & y_4 \end{array}\right] \left[\begin{array}{c} a_1 \\ a_2 \end{array}\right] \qquad \text{or} \qquad \left[\begin{array}{c} \mathbf{r}_1 \\ \mathbf{r}_2 \end{array}\right] = \left[\begin{array}{c} \mathbf{Y}_1 \\ \mathbf{Y}_2 \end{array}\right] \mathbf{a} \quad (8.3)$$

$$\left[\begin{array}{c} r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \\ \hline r_6 \\ r_7 \\ r_8 \\ r_9 \end{array}\right] = \left[\begin{array}{cccccc} a_2 & a_1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & a_2 & a_1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & a_2 & a_1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & a_2 & a_1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & a_2 & a_1 \\ \hline a_2 & \cdot & a_1 & \cdot & \cdot & \cdot \\ \cdot & a_2 & \cdot & a_1 & \cdot & \cdot \\ \cdot & \cdot & a_2 & \cdot & a_1 & \cdot \\ \cdot & \cdot & \cdot & a_2 & \cdot & a_1 \end{array}\right] \left[\begin{array}{c} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{array}\right] \quad \text{or} \quad \left[\begin{array}{c} \mathbf{r}_1 \\ \mathbf{r}_2 \end{array}\right] = \left[\begin{array}{c} \mathbf{A}_1 \\ \mathbf{A}_2 \end{array}\right.$$

(8.4)

A new concept embedded in (8.3) and (8.4) is that one filter can be applicable for different stretchings of the filter's time axis. One wonders, "Of all classes of filters, what subset remains appropriate for stretchings of the axes?"

8.2.1. Examples of scale-invariant filtering

When we consider all functions with vanishing gradient, we notice that the gradient vanishes whether it is represented as $(1,-1)/\Delta x$ or as $(1,0,-1)/2\Delta x$. Likewise for

the Laplacian, in one dimension or more. Likewise for the wave equation, as long as there is no viscosity and as long as the time axis and space axes are stretched by the same amount. The notion of "dip filter" seems to have no formal definition, but the idea that the spectrum should depend mainly on slope in Fourier space implies a filter that is scale-invariant. I expect the most fruitful applications to be with dip filters.

Resonance or viscosity or damping easily spoils scale-invariance. The resonant frequency of a filter shifts if we stretch the time axis. The difference equations

$$y_t - \alpha y_{t-1} = 0 \tag{8.5}$$

$$y_t - \alpha^2 y_{t-2} = 0 \tag{8.6}$$

both have the same solution $y_t = y_0 \alpha^{-t}$. One difference equation has the filter $(1, -\alpha)$, while the other has the filter $(1, 0, -\alpha^2)$, and α is not equal to α^2. Although these operators differ, when $\alpha \approx 1$ they might provide the same general utility, say as a roughening operator in a fitting goal.

Another aspect to scale-invariance work is the presence of "parasitic" solutions, which exist but are not desired. For example, another solution to $y_t - y_{t-2} = 0$ is the one that oscillates at the Nyquist frequency.

(Viscosity does not necessarily introduce an inherent length and thereby spoil scale-invariance. The approximate frequency independence of sound absorption per wavelength typical in real rocks is a consequence of physical inhomogeneity at all scales. See for example Kjartansson's constant Q viscosity, described in IEI. Kjartansson teaches that the decaying solutions $t^{-\gamma}$ are scale-invariant. There is no "decay time" for the function $t^{-\gamma}$. Differential equations of finite order and difference equations of finite order cannot produce $t^{-\gamma}$ damping, yet we know that such damping is important in observations. It is easy to manufacture $t^{-\gamma}$ damping in Fourier space; $\exp[(-i\omega)^{\gamma+1}]$ is used. Presumably, difference equations can make reasonable approximations over a reasonable frequency range.)

8.2.2. Scale-invariance introduces more fitting equations

The fitting goals (8.3) and (8.4) have about double the usual number of fitting equations. Scale-invariance *introduces extra equations*. If the range of scale-invariance is wide, there will be more equations. Now we begin to see the big picture.

1. Refining a model mesh improves accuracy.
2. Refining a model mesh makes empty bins.

3. Empty bins spoil analysis.
4. If there are not too many empty bins we can find a PEF.
5. With a PEF we can fill the empty bins.
6. To get the PEF and to fill bins we need enough equations.
7. Scale-invariance introduces more equations.

An example of these concepts is shown in Figure 8.2. Additionally, when we have a PEF, often we still cannot find missing data because conjugate-direction iterations do not converge fast enough (to fill large holes). Multiscale convolutions should converge quicker because they are like mesh-refinement, which is quick. An example of these concepts is shown in Figure 8.3.

8.2.3. Coding the multiscale filter operator

Equation (8.3) shows an example where the first output signal is the ordinary one and the second output signal used a filter interlaced with zeros. We prepare subroutines that allow for more output signals, each with its own filter interlace parameter

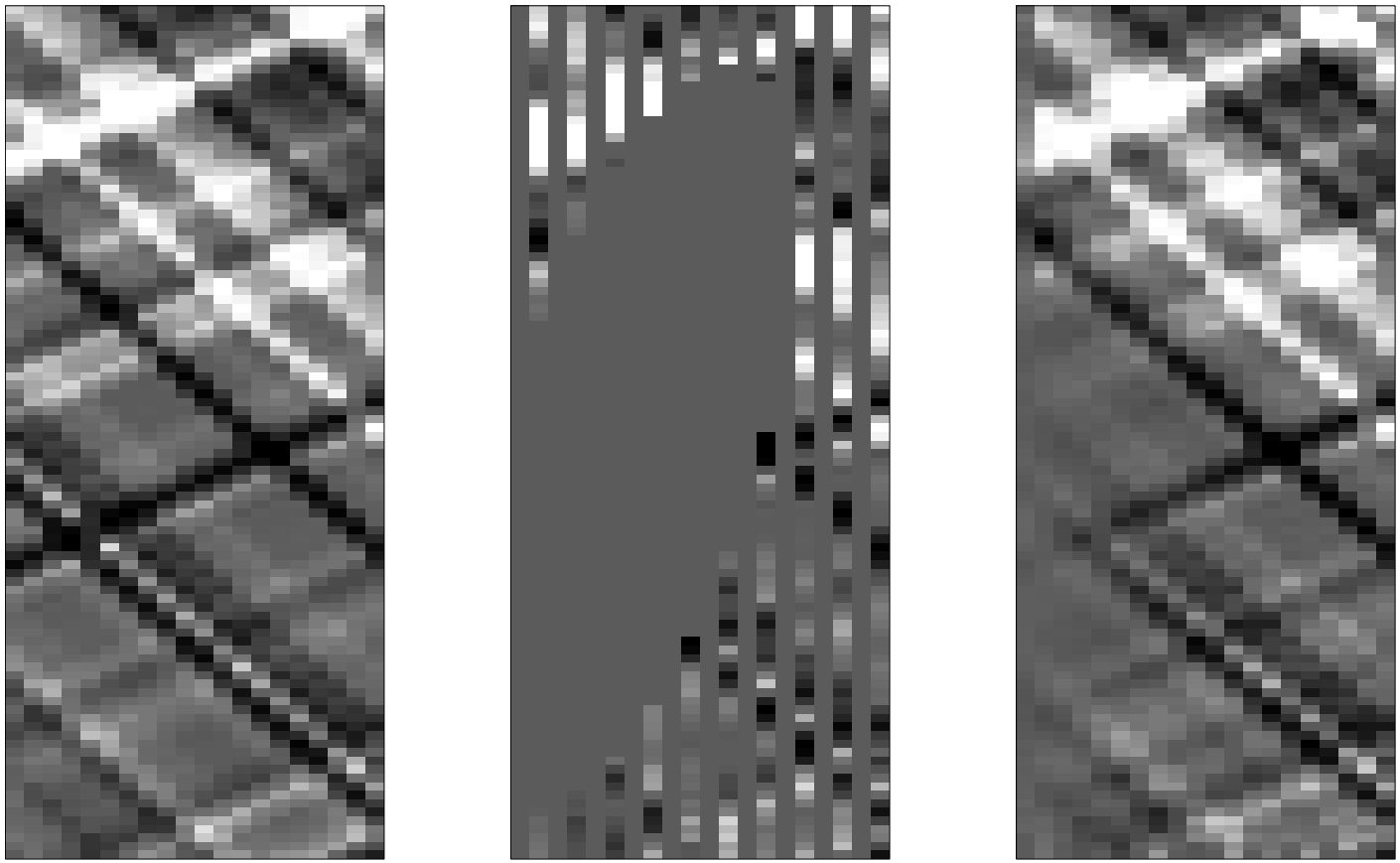
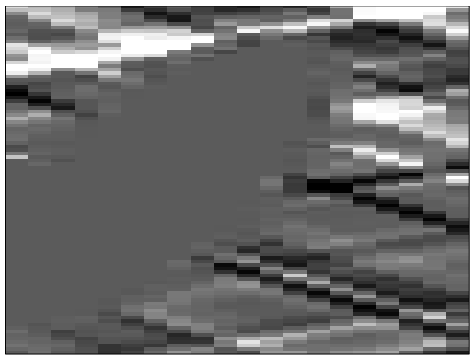
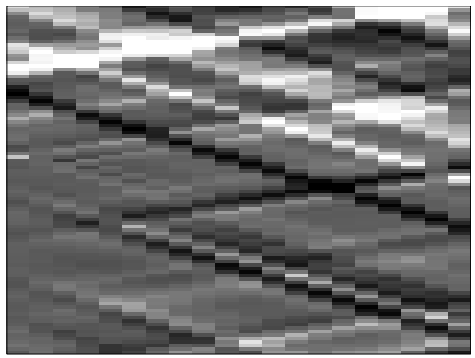


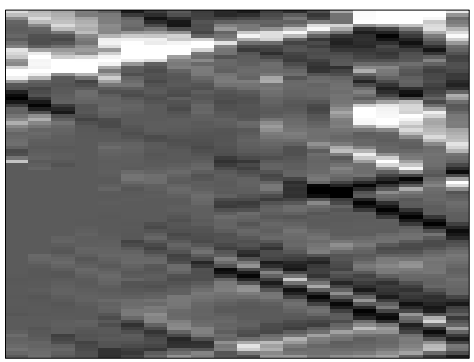
Figure 8.2: Overcoming aliasing with multiscale fitting. lal-mshole90 [ER]



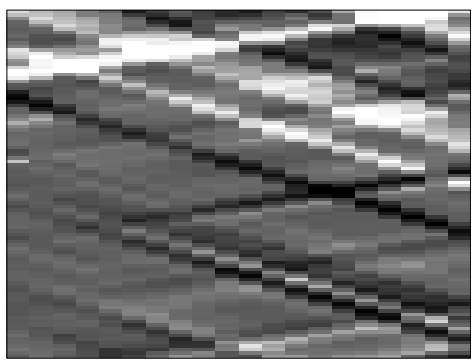
single_scale:_iter=1



single_scale:_iter=30



multiscale:___iter=1



multiscale:___iter=30

Figure 8.3: Large holes are filled faster with multiscale operators. lal-msiter90 [ER]

given in the table `jump(ns)`. Each entry in the jump table corresponds to a particular scaling of a filter axis. The number of output signals is `ns` and the number of zeros interlaced between filter points for the `j`-th signal is `jump(j)-1`.

The multiscale helix filter is defined in module `mshelix` /prog:mshelix, analogous to the single-scale module `helix` /prog:helix. A new function `onescale` extracts our usual helix filter of one particular scale from the multiscale filter.

mshelix We create a multscale helix with module `createmshelixmod` /prog:createmshe An expanded scale helix filter is like an ordinary helix filter except that the lags are scaled according to a `jump`. createmshelixmod

First we examine code for estimating a prediction-error filter that is applicable at many scales. We simply invoke the usual filter operator `hconest` /prog:hconest for each scale. mshconest

The multiscale prediction-error filter finding subroutine is nearly identical to the usual subroutine `find_pef()` /prog:pef. (That routine cleverly ignores missing data while estimating a PEF.) To easily extend `pef` to multiscale filters we replace its call to the ordinary helix filter module `hconest` /prog:hconest by a call to `mshconest`. mspef The purpose of `pack(dd,.true.)` is to produce the one-dimensional

```
module mshelix {                                    # multiscale helix filter type
  use helix
  type msfilter {
    real,    dimension( :),    pointer :: flt   # (nh) filter coefficients
    integer, dimension( :, :), pointer :: lag   # (nh,ns) filter (lags,scales)
    logical, dimension( :, :), pointer :: mis   # (nd,ns) boundary conditions
  }
contains
  subroutine msallocate( msaa, nh, ns) {
    type( msfilter)   :: msaa
    integer           :: nh, ns
    allocate( msaa%flt( nh), msaa%lag( nh, ns))
    msaa%flt = 0.;  nullify( msaa%mis)
    }
  subroutine msdeallocate( msaa) {
    type( msfilter) :: msaa
    deallocate( msaa%flt, msaa%lag)
    if( associated( msaa%mis)) deallocate( msaa%mis)
    }
  subroutine onescale( i, msaa, aa) {      # Extract single-scale filter.
    integer, intent (in) :: i
    type( filter)        :: aa
    type( msfilter)      :: msaa
    aa%flt => msaa%flt
    aa%lag => msaa%lag( :, i)
    if( associated( msaa%mis))
        aa%mis => msaa%mis( :, i)
    else
        nullify( aa%mis)
    }
}
```

Back

```
module createmshelixmod {          # Create multiscale helix filter lags and mis
use mshelix
use createhelixmod
use bound
contains
  function createmshelix( nd, center, gap, jump, na)  result( msaa) {
    type( msfilter)                    :: msaa    # needed by mshelicon.
    integer, dimension(:), intent(in) :: nd, na  # data and filter axes
    integer, dimension(:), intent(in) :: center  # normally (na1/2,na2/2,...,1)
    integer, dimension(:), intent(in) :: gap     # normally ( 0,   0,  0,...,0)
    integer, dimension(:), intent(in) :: jump    # jump(ns) stretch scales
    type( filter)                      :: aa
    integer                            :: is, ns, nh, n123
    aa = createhelix( nd, center, gap, na)
    ns = size( jump);  nh = size( aa%lag);  n123 = product( nd)
    call msallocate( msaa, nh, ns)
    do is = 1, ns
        msaa%lag(:,is) = aa%lag(:)*jump(is)      # set lags for expanded scale
    call deallocatehelix( aa)
    allocate( msaa%mis( n123, ns))
    do is = 1, ns {                                  # for all scales
     call onescale( is, msaa, aa);  nullify( aa%mis)    # extract a filter
     call boundn( nd, nd, na*jump(is), aa)              # set up its boundaries
     msaa%mis( :, is) = aa%mis;  deallocate( aa%mis)    # save them
    }
  }
}
```

Back

```
module mshconest {      # multi-scale helix convolution, adjoint is the filter.
  use mshelix
  use hconest
  use helix
  integer, private                    :: nx, ns
  real,    dimension(:),     pointer :: x
  type( msfilter)                     :: msaa
#% _init( x, msaa)
   nx = size( x);   ns = size( msaa%lag, 2)
#% _lop( a(:), y(nx,ns))
    integer  :: is, stat1
    type (filter) :: aa
    do is = 1, ns {
        call onescale (is, msaa, aa)
        call hconest_init( x, aa)
        stat1 = hconest_lop( adj, .true., a, y(:,is))
    }
}
```

Back

```
module mspef {         # Find multi-scale prediction-error filter (helix magic)
  use mshconest
  use cgstep_mod
  use solver_smp_mod
contains
  subroutine find_pef( yy, aa, niter) {
    integer,             intent( in)             :: niter
    real, dimension( :), pointer                 :: yy
    type( msfilter)                              :: aa
    integer                                      :: is
    real, allocatable                 :: dd(:,:),ee(:)
    allocate(dd( size( yy), size( aa%lag, 2)))
    allocate(ee(size(dd)))
    do is = 1, size( dd, 2)
       dd( :,is) = -yy
                ee=reshape( dd,(/size(dd)/))
    call mshconest_init( yy, aa)
    call solver_smp(m=aa%flt, d=ee, Fop=mshconest_lop, step-
per=cgstep, niter=niter, m0=aa%flt)
    call cgstep_close()
                deallocate(dd,ee)
    }
}
```

Back

array expected by our solver routines.

Similar code applies to the operator in (8.4) which is needed for missing data problems. This is like `mshconest` /prog:mshconest except the adjoint is not the filter but the input. mshelicon The multiscale missing-data module `msmis2` is just like the usual missing-data module `mis2` /prog:mis2 except that the filtering is done with the multiscale filter `mshelicon` /prog:mshelicon. msmis2

8.3. References

Canales, L.L., 1984, Random noise reduction: 54th Ann. Internat. Mtg., Soc. Explor. Geophys., Expanded Abstracts, 525-527.

Rothman, D., 1985, Nonlinear inversion, statistical mechanics, and residual statics estimation: Geophysics, **50**, 2784-2798

Spitz, S., 1991, Seismic trace interpolation in the F-X domain: Geophysics, **56**, 785-794.

```
module mshelicon {                                      # Multi-scale convolution
  use mshelix
  use helicon
  integer             :: nx, ns
  type( msfilter)     :: msaa
#%  _init (nx, ns, msaa)
#%  _lop ( xx( nx),  yy( nx, ns))
  integer :: is, stat1
  type (filter) :: aa
  do is = 1, ns {
     call onescale( is, msaa, aa)
     call helicon_init( aa)
     stat1 = helicon_lop( adj, .true., xx, yy(:,is))
     }
}
```

Back

```
module msmis2 {                         # multi-scale missing data interpolation
  use mshelicon
  use cgstep_mod
  use mask1
  use solver_smp_mod
contains
  subroutine mis1( niter, nx, ns, xx, aa, known) {
    integer,                intent( in)     :: niter, nx, ns
    logical, dimension( :), intent( in)     :: known
    type( msfilter),        intent( in)     :: aa
    real,    dimension( :), intent( in out) :: xx
    real,    dimension( nx*ns)              :: dd
    logical, dimension(:), pointer :: kk
    dd = 0.
    allocate(kk(size(known))); kk=.not. known
    call mask1_init(kk)
    call mshelicon_init( nx,ns, aa)
    call solver_smp(m=xx, d=dd, Fop=mshelicon_lop, step-
per=cgstep, niter=niter, Jop=mask1_lop, m0=xx)
    call cgstep_close()
  }
}
```

Back

Chapter 9

Nonstationarity: patching

There are many reasons for cutting data planes or image planes into overlapping pieces (patches), operating on the pieces, and then putting them back together again, as depicted in Figure 9.1. The earth's dip varies with lateral location and depth. The

dip spectrum and spatial spectrum thus also varies. The dip itself is the essence of almost all earth mapping, and its spectrum plays an important role in the estimation any earth properties. In statistical estimation theory, the word to describe changing statistical properties is “nonstationary”.

We begin this chapter with basic patching concepts along with supporting utility code. The language of this chapter, *patch*, *overlap*, *window*, *wall*, is two-dimensional, but it may as well be three-dimensional, *cube*, *subcube*, *brick*, or one-dimensional, *line*, *interval*. We sometimes use the language of windows on a wall. But since we usually want to have overlapping windows, better imagery would be to say we assemble a quilt from patches.

The codes are designed to work in any number of dimensions. After developing the infrastructure, we examine some two-dimensional, time- and space-variable applications: adaptive steep-dip rejection, noise reduction by prediction, and segregation of signals and noises.

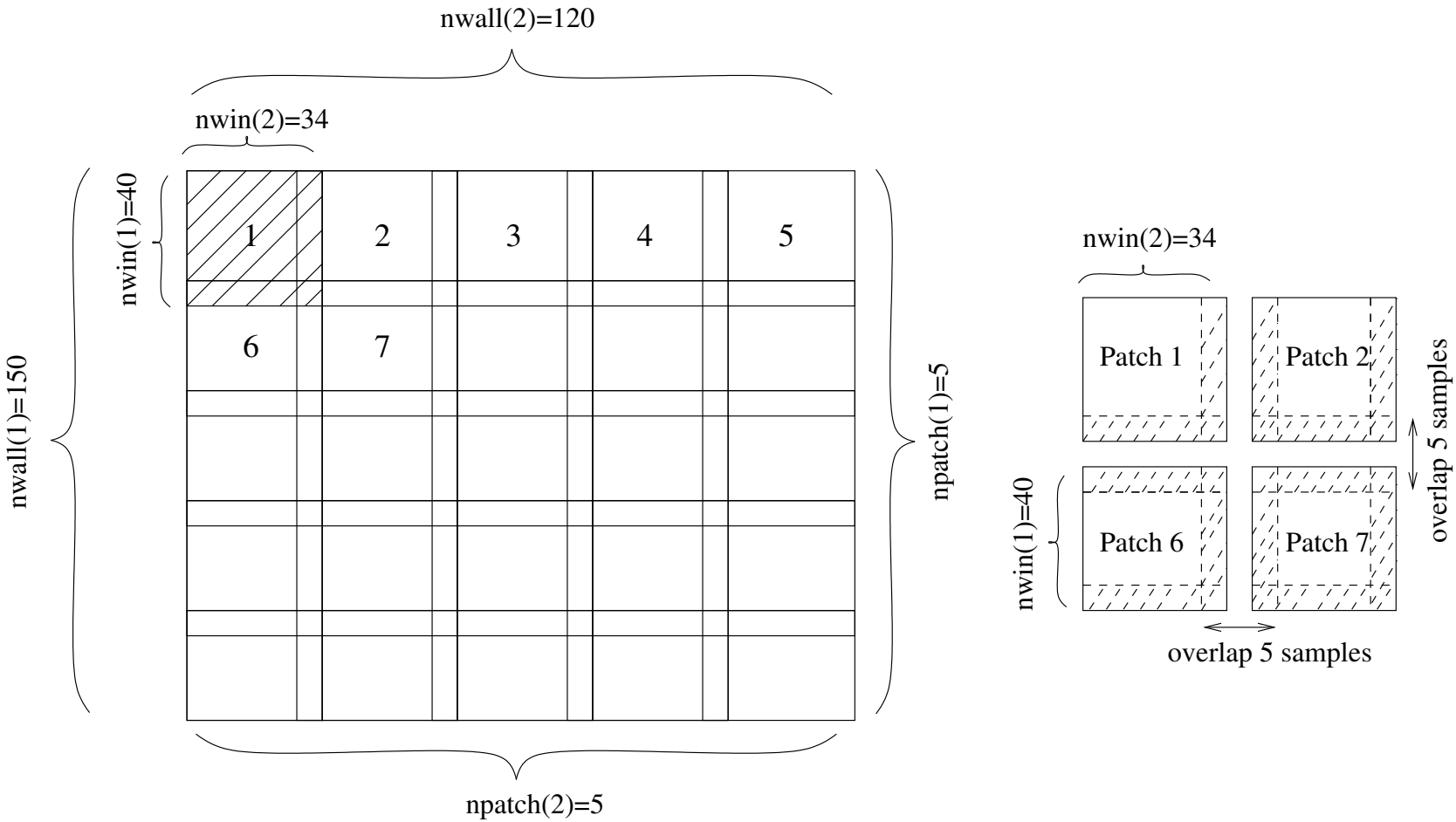


Figure 9.1: Decomposing a wall of information into windows (also called patches). Left is an example of a 2-D space input to module `patch`. Right shows a close-up of the output (top left corner). pch-antoine [NR]

9.1. PATCHING TECHNOLOGY

A plane of information, either data or an image, say `wall(nwall1, nwall2)`, will be divided up into an array of overlapping windows each window of size `(nwind1,nwind2)`. To choose the number of windows, you specify `(npatch1,npatch2)`. Overlap on the 2-axis is measured by the fraction `(nwind2*npatch2)/nwall2`. We turn to the language of F90 which allows us to discuss N-dimensional hypercubes almost as easily as two-dimensional spaces. We define an N-dimensional volume (like the wall) with the vector `nwall= (nwall1, nwall2, ...)`. We define subvolume size (like a 2-D window) with the vector `nwind=(nwind1, nwind2, ...)`. The number of subvolumes on each axis is `npatch=(npatch1, npatch2, ...)`. The operator `patch` /prog:patch simply grabs one patch from the wall, or when used in adjoint form, it puts the patch back on the wall. The number of patches on the wall is `product(npatch)`. Getting and putting all the patches is shown later in module `patching` /prog:patching.

The i-th patch is denoted by the scalar counter `ipatch`. Typical patch extraction begins by taking `ipatch`, a fortran linear index, and converting it to a multidimensional subscript `jj` each component of which is less than `npatch`. The patches cover all edges and corners of the given data plane (actually the hypervolume) even where

`nwall/npatch` is not an integer, even for axes whose length is not an integer number of the patch length. Where there are noninteger ratios, the spacing of patches is slightly uneven, but we'll see later that it is easy to reassemble seamlessly the full plane from the patches, so the unevenness does not matter. You might wish to review the utilities `line2cart` and `cart2line` /prog:cartesian which convert between multidimensional array subscripts and the linear memory subscript before looking at the patch extraction-putback code: patch The cartesian vector `jj` points to the beginning of a patch, where on the wall the (1,1,..) coordinate of the patch lies. Obviously this begins at the beginning edge of the wall. Then we pick `jj` so that the last patch on any axis has its last point exactly abutting the end of the axis. The formula for doing this would divide by zero for a wall with only one patch on it. This case arises legitimately where an axis has length one. Thus we handle the case `npatch=1` by abutting the patch to the beginning of the wall and forgetting about its end. As in any code mixing integers with floats, to guard against having a floating-point number, say 99.9999, rounding down to 99 instead of up to 100, the rule is to always add .5 to a floating point number the moment before converting it to an integer. Now we are ready to sweep a window to or from the wall. The number of points in a window is `size(wind)` or equivalently `product(nwind)`. Figure 9.2 shows an example

```
module patch {                      # N-dimensional patch extract and putback
use cartesian
integer, private                 :: ipatch         # count till product(npatch)
integer, dimension (:), pointer :: npatch,  nwind,  nwall
                # overlap where    npatch * nwind > nwall
#% _init( npatch, nwall, nwind)
ipatch = 1
#% _lop ( wall, wind)
  integer, dimension(:),allocatable :: ii, jj        # Ndim=size(npatch)
  integer                            :: i, j, shift
                allocate(ii(size(npatch)),jj(size(npatch)))
  call line2cart(  npatch, ipatch, jj )  # (j1,j2) = (1,1) to (npatch1,npatch2)
              # round jj to shift end of last patch flush against the far wall.
  where( npatch==1) { jj = 1   }
  elsewhere         { jj = 1.5 +( nwall - nwind)*( jj - 1.)/( npatch - 1.)}
  call cart2line( nwall,   jj, shift)    # jj is where the patch begins.
  shift -= 1                             #
  do i  = 1, size( wind) {               # sweep through a window
     call line2cart(  nwind, i, ii)      #   ii(i)   is (i1,i2) in window
     call cart2line(  nwall,    ii, j)   # j(ii)     linear subscript in window
     if( adj)
        wall( j + shift) += wind( i)
     else
        wind( i)         += wall( j + shift)
  }
        deallocate(ii,jj)
#% _close
ipatch = ipatch + 1
}
```

Back

with five nonoverlapping patches on the 1-axis and many overlapping patches on the 2-axis.

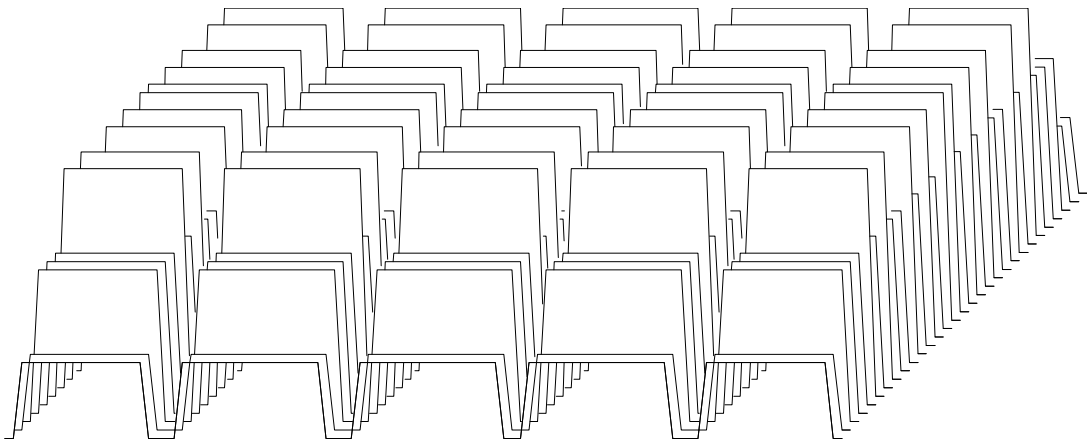


Figure 9.2: A plane of identical values after patches have been cut and then added back. Results are shown for `nwall=(100,30)`, `nwind=(17,6)`, `npatch=(5,11)`. For these parameters, there is gapping on the horizontal axis and overlap on the depth axis. pch-parcel90 [ER]

9.1.1. Weighting and reconstructing

The adjoint of extracting all the patches is adding them back. Because of the overlaps, the adjoint is not the inverse. In many applications, inverse patching is re-

quired; i.e. patching things back together seamlessly. This can be done with weighting functions. You can have any weighting function you wish and I will provide you the patching reconstruction operator $\tilde{\mathbf{I}}_p$ in

$$\tilde{\mathbf{d}} \quad = \quad [\mathbf{W}_{\text{wall}}\mathbf{P}'\mathbf{W}_{\text{wind}}\mathbf{P}]\mathbf{d} \quad = \quad \tilde{\mathbf{I}}_p\,\mathbf{d} \tag{9.1}$$

where $\mathbf{d}$ is your initial data, $\tilde{\mathbf{d}}$ is the reconstructed data, $\mathbf{P}$ is the patching operator, $\mathbf{P}'$ is adjoint patching (adding the patches). $\mathbf{W}_{\text{wind}}$ is your chosen weighting function in the window, and $\mathbf{W}_{\text{wall}}$ is the weighting function for the whole wall. You specify any $\mathbf{W}_{\text{wind}}$ you like, and module `mkwallwt` below builds the weighting function $\mathbf{W}_{\text{wall}}$ that you need to apply to your wall of reconstructed data, so it will undo the nasty effects of the overlap of windows and the shape of your window-weighting function. You do not need to change your window weighting function when you increase or decrease the amount of overlap between windows because $\mathbf{W}_{\text{wall}}$ takes care of it. The method is to use adjoint `patch` /prog:patch to add the weights of each window onto the wall and finally to invert the sum wherever it is non-zero. (You lose data wherever the sum is zero). mkwallwt

No matrices are needed to show that this method succeeds, because data values are never mixed with one another. An equation for any reconstructed data

```
module mkwallwt {                  # N-dimensional wall weights
use patch
contains
  subroutine wallwtn( n, nwall, nwind, windwt, wallwt) {
    integer, dimension( :), pointer      :: n, nwall, nwind
    real,    dimension( :), intent( in)  :: windwt
    real,    dimension( :), intent( out) :: wallwt
    integer                              :: ipatch, stat
    wallwt = 0.
    call patch_init( n, nwall, nwind)
    do ipatch = 1, product( n) {
       stat = patch_lop( .true., .true., wallwt, windwt)
       call patch_close ()
    }
    where( wallwt /= 0.) wallwt = 1. / wallwt
  }
}
```

Back

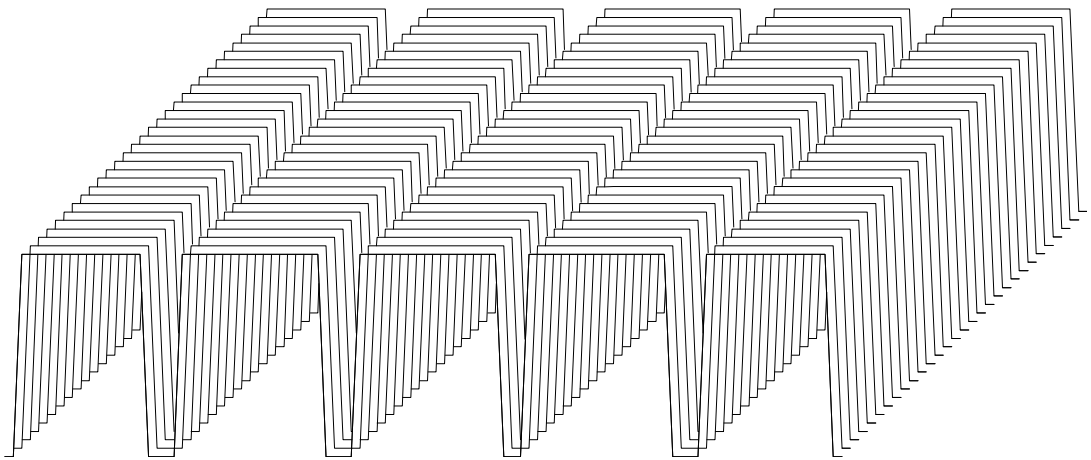
value $\tilde{d}$ as a function of the original value d and the weights w_i that hit d is $\tilde{d} = (\sum_i w_i d)/\sum_i w_i = d$. Thus, our process is simply a "partition of unity."

To demonstrate the program, I made a random weighting function to use in each window with positive random numbers. The general strategy allows us to use different weights in different windows. That flexibility adds clutter, however, so here we simply use the same weighting function in each window.

The operator $\tilde{\mathbf{I}}_p$ is called "idempotent." The word "idempotent" means "self-power," because for any N, $0^N = 0$ and $1^N = 1$, thus the numbers 0 and 1 share the property that raised to any power they remain themselves. Likewise, the patching reconstruction operator multiplies every data value by either one or zero. Figure 9.3 shows the result obtained when a plane of identical constant values **d** is passed into the patching reconstruction operator $\tilde{\mathbf{I}}_p$. The result is constant on the 2-axis, which confirms that there is adequate sampling on the 2-axis, and although the weighting function is made of random numbers, all trace of random numbers has disappeared from the output. On the 1-axis the output is constant, except for being zero in gaps, because the windows do not overlap on the 1-axis.

Module `patching` assists in reusing the patching technique. It takes a linear operator **F**. as its argument and applies it in patches. Mathematically, this is

Figure 9.3: A plane of identical values passed through the idempotent patching reconstruction operator. Results are shown for the same parameters as Figure 9.2. pch-idempatch90 [ER]



```
module patching {                    # Apply existing operator in all patches
  use patch
  use mkwallwt
contains
  subroutine patchingn( oper, modl, data, npatch, nwall, nwind, windwt) {
    interface {
       integer function oper(adj, add, modl, data) {
          logical, intent (in) :: adj, add
          real, dimension( :)  :: modl, data
          }
       }
    real,    dimension( :), intent( in)  :: windwt, modl
    real,    dimension( :), intent( out) :: data
    integer, dimension( :), pointer      :: npatch, nwall, nwind
    real,    dimension( size( modl))     :: wallwt
    real,    dimension( size( windwt))   :: winmodl, windata
    integer                              :: i, stat1, stat2
    data = 0.
    call patch_init( npatch, nwall, nwind)
    do i = 1, product( npatch) {
       stat1 = patch_lop( .false., .false., modl, winmodl)
       stat2 = oper(      .false., .false., winmodl, windata)
       stat1 = patch_lop( .true. , .true. , data,  windwt * windata)
       call patch_close ()
       }
    call wallwtn( npatch, nwall, nwind, windwt, wallwt);  data = data * wallwt
    }
}
```

Back

$[\mathbf{W}_{\text{wall}}\mathbf{P}'\mathbf{W}_{\text{wind}}\mathbf{F}\mathbf{P}]\mathbf{d}$. It is assumed that the input and output sizes for the operator `oper` are equal. patching

9.1.2. 2-D filtering in patches

A way to do time- and space-variable filtering is to do invariant filtering within each patch. Typically, we apply a filter, say $\mathbf{F}_p$, in each patch. The composite operator, filtering in patches, $\tilde{\mathbf{F}}$, is given by

$$\tilde{\mathbf{d}} \quad = \quad [\mathbf{W}_{\text{wall}}\mathbf{P}'\mathbf{W}_{\text{wind}}\mathbf{F}_p\mathbf{P}]\,\mathbf{d} \quad = \quad \tilde{\mathbf{F}}\,\mathbf{d} \tag{9.2}$$

I built a triangular weighting routine `tentn()` that tapers from the center of the patch of the filter's *outputs* towards the edges. Accomplishing this weighting is complicated by (1) the constraint that the filter must not move off the edge of the input patch and (2) the alignment of the input and the output. The layout for prediction-error filters is shown in Figure 9.4. We need a weighting function that vanishes where the filter has no outputs. The amplitude of the weighting function is not very important because we have learned how to put signals back together properly for arbitrary weighting functions. We can use any pyramidal or tent-like shape that

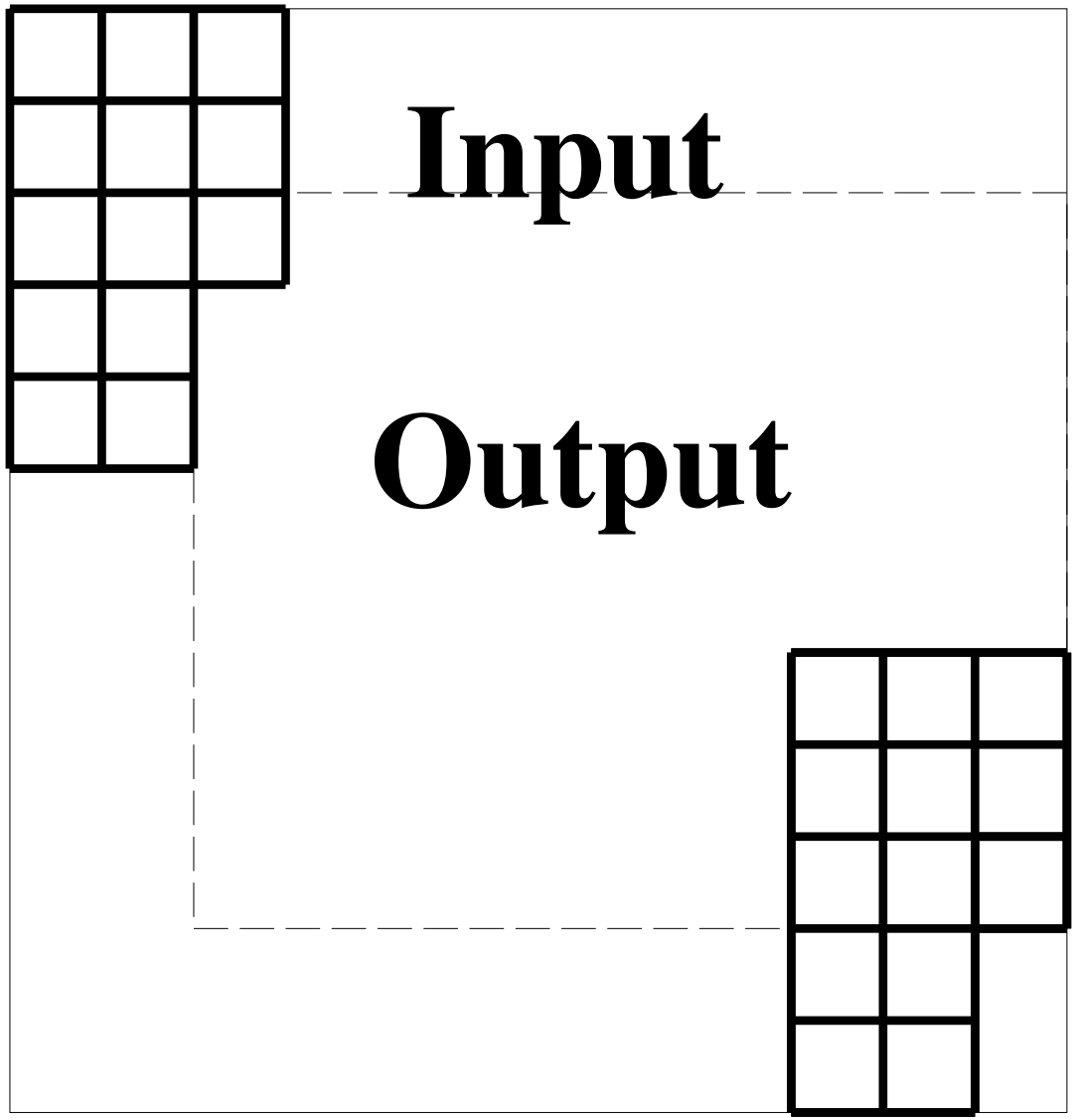


Figure 9.4: Domain of inputs and outputs of a two-dimensional prediction-error filter. pch-rabdomain [NR]

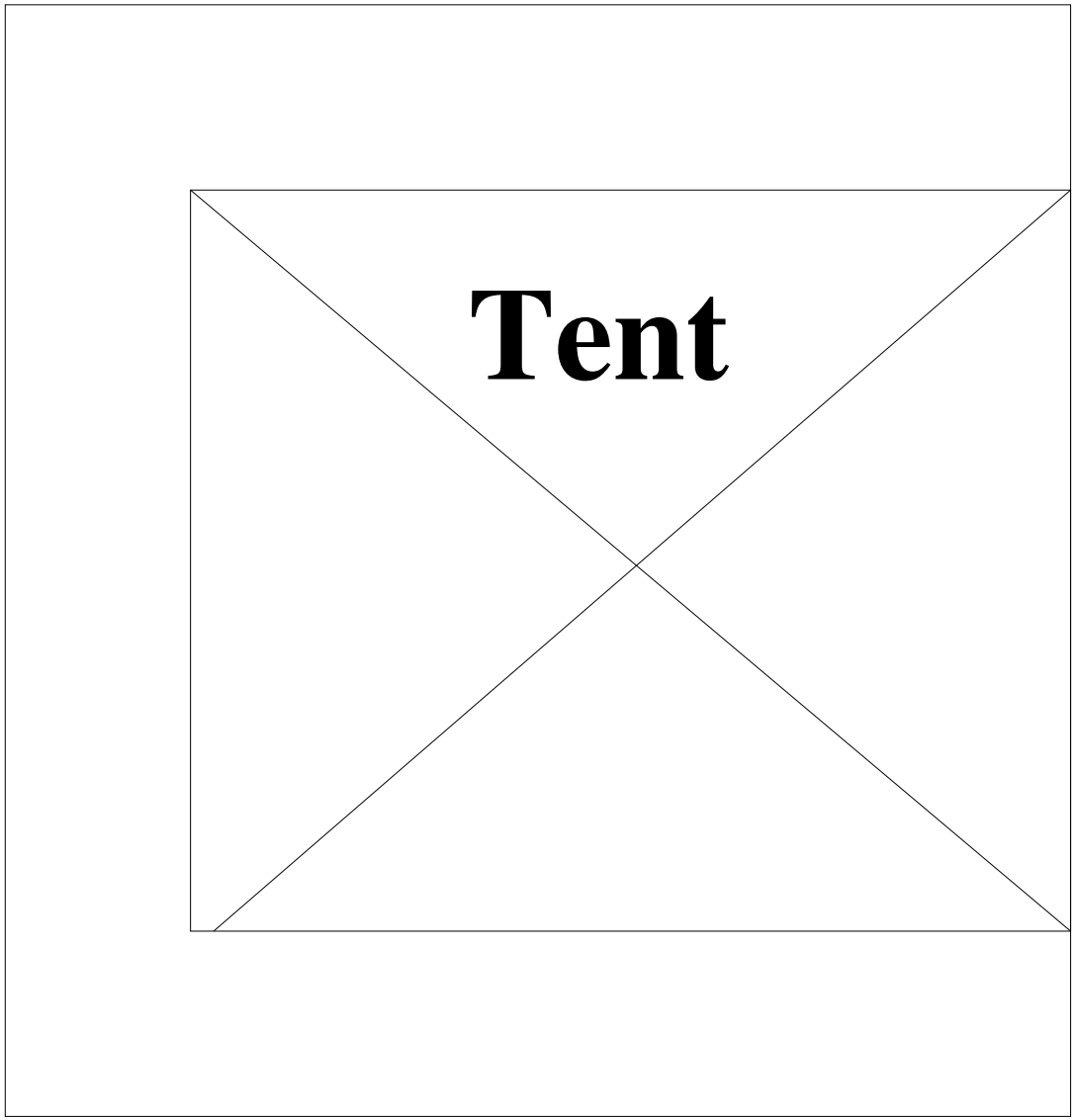


Figure 9.5: Placement of tent-like weighting function in the space of filter inputs and outputs. pch-rabtent [NR]

```
module tent {  # triangle tent weights in N dimensions
use cartesian
contains
subroutine tentn ( nwind, center, a, windwt) {
  integer, dimension (:), intent (in)  :: nwind, center, a
  real,    dimension (:), intent (out) :: windwt
  integer, dimension( size( nwind))    :: start, end, x
  real,    dimension( size( nwind))    :: mid, wid
  integer                              :: i
  start= 1+a-center;     end= nwind-center+1         # a is beginning of tent
  mid=  (end+start)/2.;  wid= (end-start+1.)/2.
  do i = 1, size( windwt) {
       call line2cart(  nwind, i, x)
       if( all( x >= start) .and. all( x <= end))
           windwt( i) = product( max( 0., 1. - abs((x-mid)/wid)))
       else
           windwt( i) = 0.
       }
  }
}
```

Back

drops to zero outside the domain of the filter output. The job is done by subroutine `tentn()`. A new parameter needed by `tentn` is `a`, the coordinate of the beginning of the tent. tent

In applications where triangle weights are needed on the *inputs* (or where we can work on a patch without having interference with edges), we can get "triangle tent" weights from `tentn()` if we set filter dimensions and lags to unity, as shown in Figure 9.6.

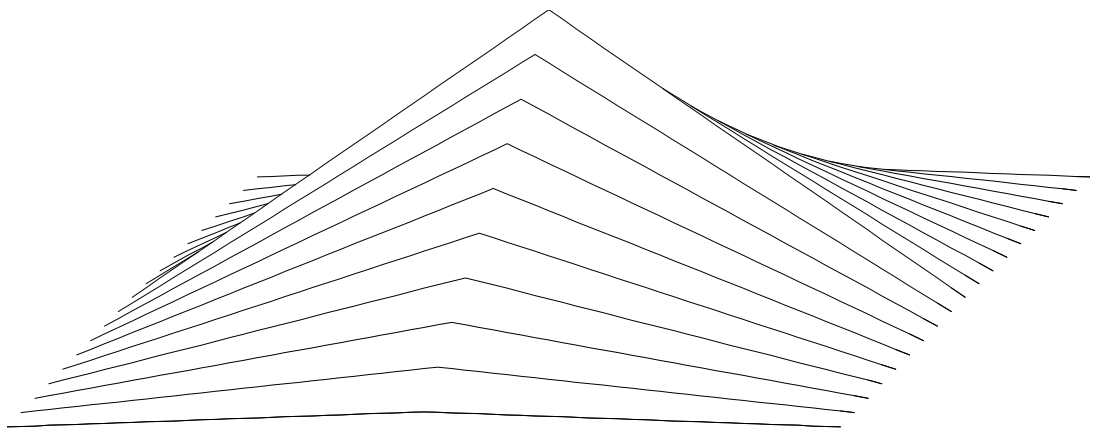


Figure 9.6: Window weights from `tentn()` with `nwind=(61,19), center=(31,1), a=(1,1)` . pch-windwt90 [ER]

Triangle weighting functions can sum to a constant if the spacing is such that the midpoint of one triangle is at the beginning of the next. I imagined in two dimensions that something similar would happen with shapes like Egyptian pyramids

of Cheops, $2 - |x - y| + |x + y|$. Instead, the equation $(1 - |x|)(1 - |y|)$ which has the tent-like shape shown in Figure 9.6 adds up to the constant flat top shown in Figure 9.7. (To add interest to Figure 9.7, I separated the windows by a little more than the precise matching distance.) In practice we may chose window shapes and overlaps for reasons other than the constancy of the sum of weights, because `mkwallwt` /prog:mkwallwt accounts for that.

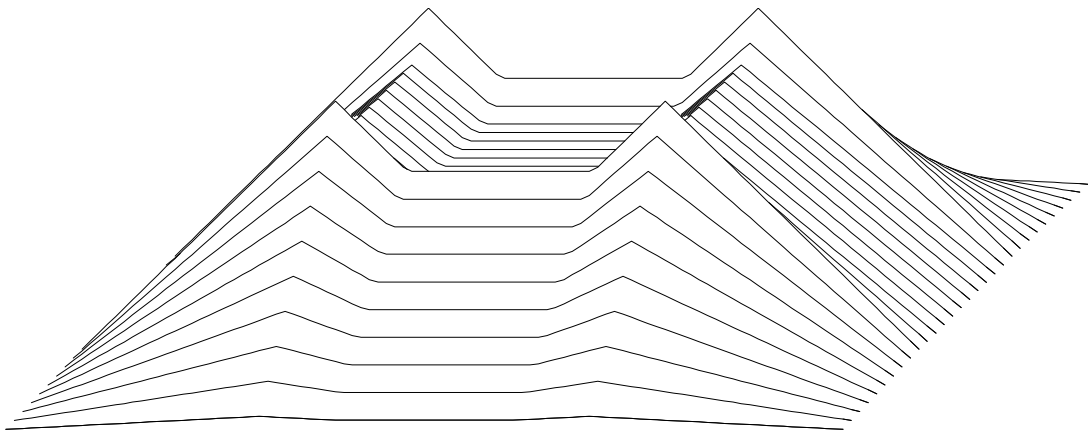


Figure 9.7: (Inverse) wall weights with `n1=100, w1=61, k1=2, n2=30, w2=19, k2=2` pch-wallwt90 [ER]

Finally is an example of filtering a plane of uniform constants with an impulse function. The impulse function is surrounded by zeros, so the filter output patches are smaller than the input patches back in Figure 9.3. Here in Figure 9.8, both axes need more window density.

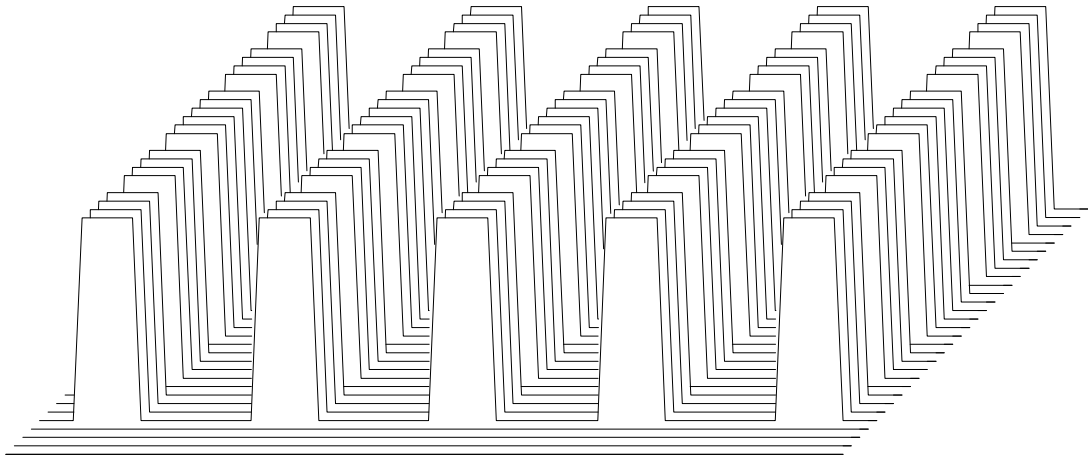


Figure 9.8: Filtering in patches with the same parameters as in Figures 9.2 and 9.3. Additionally, the filter parameters are `a1=11 a2=5 lag1=6 lag2=1` . Thus, windows are centered on the 1-axis and pushed back out the 2-axis. pch-cinloip90 [ER]

9.1.3. Designing a separate filter for each patch

Recall the prediction-error filter subroutine `find_pef()` /prog:pef. Given a data plane, this subroutine finds a filter that tends to whiten the spectrum of that data plane. The output is white residual. Now suppose we have a data plane where the dip spectrum is changing from place to place. Here it is natural to apply subroutine `find_pef()` in local patches. This is done by subroutine `find_lopef()`. The output of this subroutine is an array of helix-type filters, which can be used, for example,

```
module lopef {            # Local PEF estimation in patches.
  use patch               # Estimate a vector of filters, one for each patch.
  use misinput
  use pef
contains
  subroutine find_lopef( wall, aa, npatch, nwall, nwind, mask) {
    optional                                  :: mask
    integer,       dimension(:), pointer      :: npatch, nwall, nwind
    real,          dimension(:), intent( in) :: wall, mask
    type( filter), dimension(:)               :: aa
    real,          dimension(:), pointer      :: windata, winmask
    integer                                   :: i, stat
                       allocate( windata( product( nwind)))   # a patch
    if( present( mask)) allocate( winmask( product( nwind)))   # missing inputs
    call patch_init( npatch, nwall, nwind)
    do i = 1, product( npatch) {                               # do all patches
       stat = patch_lop( .false., .false., wall, windata)      # get a patch
       if( present( mask)) {
           stat = patch_lop( .false., .false., mask, winmask)
           call find_mask( (winmask /= 0.), aa (i))             # missing data
           }
       if( count(.not.aa(i)%mis) > size(aa(i)%lag))             # enuf eqns?
          call find_pef( windata, aa(i), niter=size( aa(i)%lag)) # find PEF
       else if( i > 1)
          aa(i)%flt = aa(i-1)%flt                               # use last PEF
       call patch_close()
       }
    deallocate( windata)
    if( present( mask))  deallocate( winmask)
    }
}
#       if( size(aa(i)%mis) - count(aa(i)%mis) > size(aa(i)%lag)) # enuf eqns?
```

Back

```
# successive invocations apply successive filters from a vector.
#                       (will fail a dot product test?  Oft used with patching.)
module loconvol {
use helicon
integer, private  :: i
type( filter), dimension(:), pointer :: aa
#% _init( aa)
i = 0
#% _lop( xx, yy)
 integer stat1;   i = i + 1
 call helicon_init( aa( i))
 stat1 = helicon_lop( adj, .false., xx, yy)
}
```

Back

in a local convolution operator `loconvol` /prog:loconvol. lopef We notice that when a patch has fewer regression equations than the filter has coefficients, then the filter is taken to be that of the previous patch. loconvol

9.1.4. Triangular patches

I have been running patching code for several years and my first general comment is that realistic applications often call for patches of different sizes and different shapes. (Tutorial, non-interactive Fortran code is poorly suited to this need.) Raw seismic data in particular seems more suited to triangular shapes. It is worth noting that the basic concepts in this chapter have ready extension to other shapes. For example, a rectangular shape could be duplicated into two identical patches; then data in one could be zeroed above the diagonal and in the other below; you would have to allow, of course, for overlap the size of the filter. Module `pef` /prog:pef automatically ignores the zeroed portion of the triangle, and it is irrelevant what `mis2()` /prog:mis2 does with a zeroed portion of data, if a triangular footprint of weights is designed to ignore its output.

EXERCISES:

1 Code the linear operator $\mathbf{W}_{\text{wall}}\mathbf{P}'\mathbf{W}_{\text{wind}}\mathbf{P}$ including its adjoint.

2 **Smoothing program.** Some familiar operations can be seen in a new light when done in patches. Patch the data. In each patch, find the mean value. Replace each value by the mean value. Reconstruct the wall.

3 **Smoothing while filling missing data.** This is like smoothing, but you set window weights to zero where there is no data. Because there will be a different set of weights in each window, you will need to make a simple generalization to `mkwallwt` /prog:mkwallwt.

4 **Gain control.** Divide the data into patches. Compute the square root of the sum of the squares of all the data in each patch. Divide all values in that patch by this amount. Reassemble patches.

9.2. STEEP-DIP DECON

Normally, when an autoregression filter (PEF) predicts a value at a point it uses values at earlier points. In practice, a gap may also be set between the predicted value and the earlier values. What is not normally done is to supplement the fitting signals on nearby traces. That is what we do here. We allow the prediction of a signal to include nearby signals at earlier times. The times accepted in the goal are inside a triangle of velocity less than about the water velocity. The new information allowed in the prediction is extremely valuable for water-velocity events. Wavefronts are especially predictable when we can view them along the wavefront (compared to perpendicular or at some other angle from the wavefront). It is even better on land, where noises move more slowly at irregular velocities, and are more likely to be aliased.

Using `lopef` /prog:lopef, the overall process proceeds independently in each of many overlapping windows. The most important practical aspect is the filter masks, described next.

9.2.1. Dip rejecting known-velocity waves

Consider the two-dimensional filter

$$\begin{array}{ccc} & +1 & \\ -1 & 0 & -1 \\ & +1 & \end{array} \tag{9.3}$$

When this this filter is applied to a field profile with 4 ms time sampling and 6 m trace spacing, it should perfectly extinguish 1.5 km/s water-velocity noises. Likewise, the filter

$$\begin{array}{ccc} & +1 & \\ & 0 & \\ & 0 & \\ -1 & 0 & -1 \\ & 0 & \\ & 0 & \\ & +1 & \end{array} \tag{9.4}$$

should perfectly extinguish water noise when the trace spacing is 18 m. Such noise is, of course, spatially aliased for all temporal frequencies above 1/3 of Nyquist,

but that does not matter. The filter extinguishes them perfectly anyway. Inevitably, the filter cannot both extinguish the noise and leave the signal untouched where the alias of one is equal to the other. So we expect the signal to be altered where it matches aliased noise. This simple filter does worse than that. On horizontal layers, for example, signal wavelets become filtered by $(1,0,0,-2,0,0,1)$. If the noise is overwhelming, this signal distortion is a small price to pay for eliminating it. If the noise is tiny, however, the distortion is unforgivable. In the real world, data-adaptive deconvolution is usually a good compromise.

The two-dimensional deconvolutions filters we explore here look like this:

$$
\begin{array}{ccccccccc}
x & x & x & x & x & x & x & x & x \\
x & x & x & x & x & x & x & x & x \\
. & x & x & x & x & x & x & x & . \\
. & x & x & x & x & x & x & x & . \\
. & x & x & x & x & x & x & x & . \\
. & . & x & x & x & x & x & . & . \\
. & . & x & x & x & x & x & . & . \\
. & . & x & x & x & x & x & . & . \\
. & . & . & x & x & x & . & . & . \\
. & . & . & x & x & x & . & . & . \\
. & . & . & x & x & x & . & . & . \\
. & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . \\
. & . & . & . & 1 & . & . & . & .
\end{array}
\tag{9.5}
$$

where each . denotes a zero and each x denotes a (different) adjustable filter coefficient that is chosen to minimize the power out.

You can easily imagine variations on this shape, such as a diamond instead of a triangle. I invite you to experiment with the various shapes that suggest themselves.

9.2.2. Tests of steep-dip decon on field data

Low-velocity noises on shot records are often not fully suppressed by stacking because the noises are spatially aliased. Routine field arrays are not perfect and the noise is often extremely strong. An interesting, recently-arrived data set worth testing is shown in Figure 9.9.

I scanned the forty Yilmaz and Cumro shot profiles for strong low-velocity noises and I selected six examples. To each I applied an AGC that is a slow function of time and space (triangle smoothing windows with triangle half-widths of 200 time points and 4 channels). Because my process simultaneously does both low-velocity rejection and deconvolution, I prepared more traditional 1-D deconvolutions for comparison. This is done in windows of 250 time points and 25 channels, the same filter being used for each of the 25 channels in the window. In practice, of course, considerably more thought would be given to optimal window sizes as a function of the regional nature of the data. The windows were overlapped by about

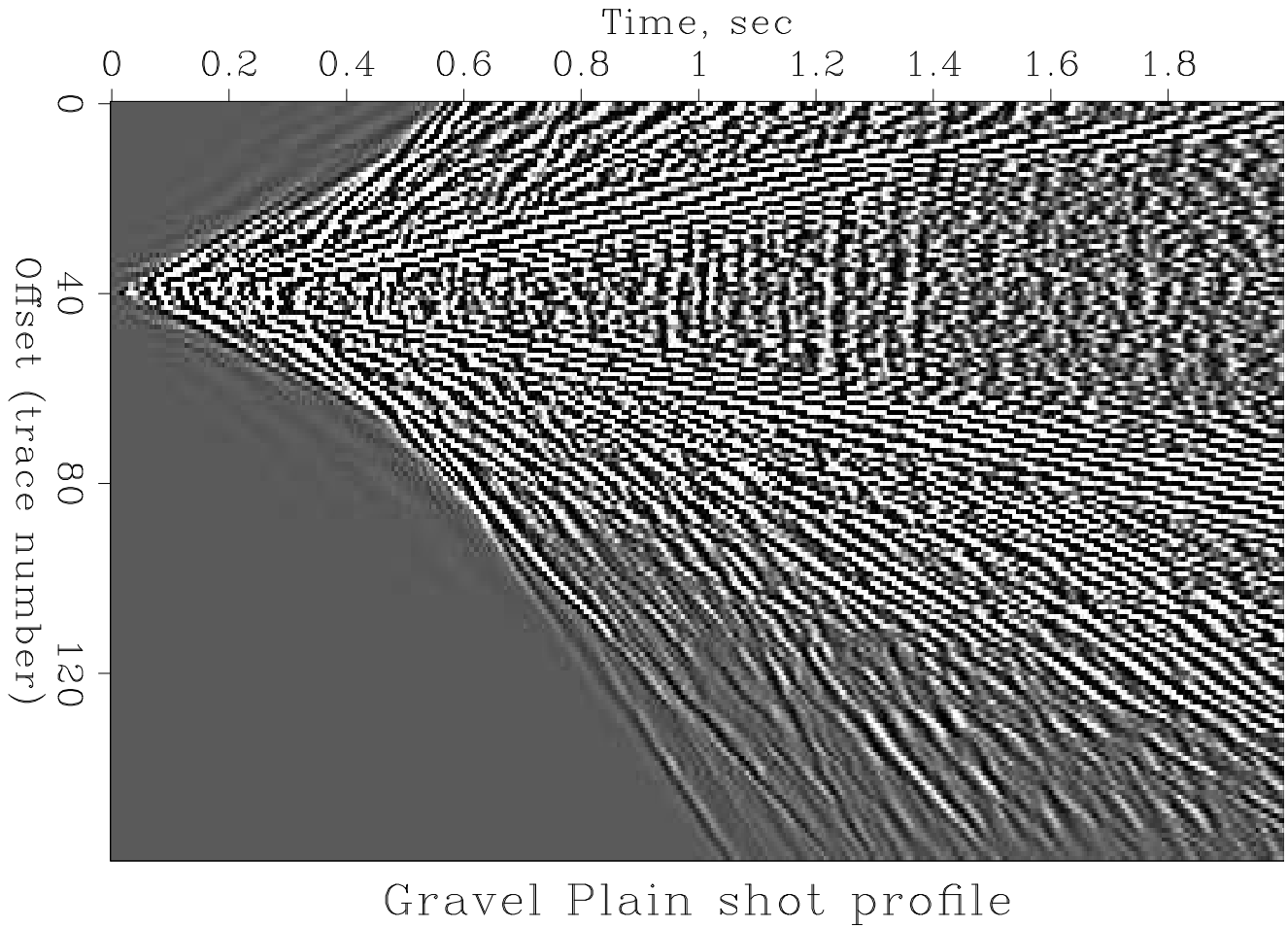


Figure 9.9: Gravel plain ground roll (Middle East) Worth testing. pch-gravel2D [ER]

50%. The same windows are used on the steep-dip deconvolution.

It turned out to be much easier than expected and on the first try I got good results on all all six field profiles tested. I have not yet tweaked the many adjustable parameters. As you inspect these deconvolved profiles from different areas of the world with different recording methods, land and marine, think about how the stacks should be improved by the deconvolution. Stanford Exploration Project report 77 (SEP-77) shows the full suite of results. Figure 9.10 is a sample of them.

Unexpectedly, results showed that 1-D deconvolution also suppresses low-velocity noises. An explanation can be that these noises are often either low-frequency or quasimonochromatic.

As a minor matter, fundamentally, my code cannot work ideally along the side boundaries because there is no output (so I replaced it by the variance scaled input). With a little extra coding, better outputs could be produced along the sides if we

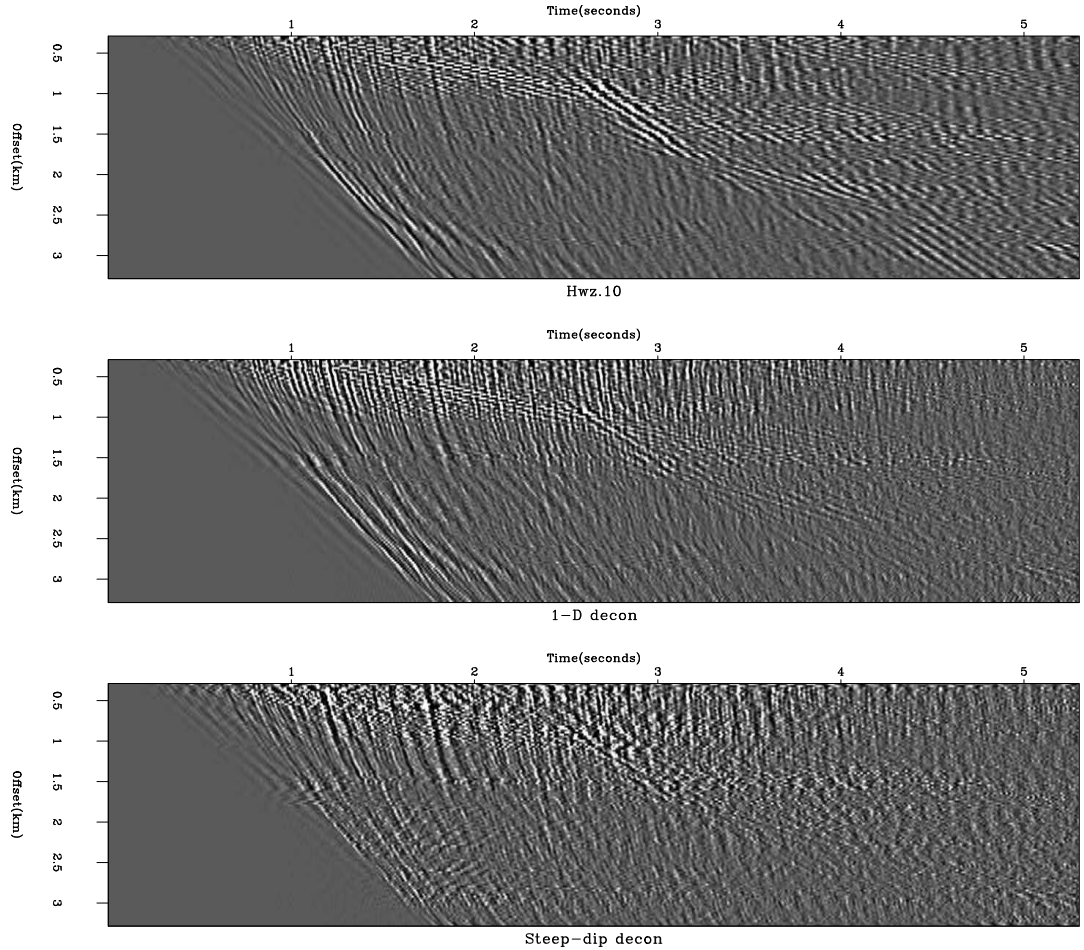


Figure 9.10: Top is a North African vibrator shot profile (Y&C #10) after AGC. Middle is gapped 1-D decon. Bottom is steep-dip decon. pch-wz.1090 [ER,M]

used spatially one-sided filters like

$$
\begin{array}{ccccc}
x & x & x & x & x \\
. & x & x & x & x \\
. & x & x & x & x \\
. & . & x & x & x \\
. & . & x & x & x \\
. & . & . & x & x \\
. & . & . & x & x \\
. & . & . & . & . \\
. & . & . & . & . \\
. & . & . & . & 1
\end{array}
\tag{9.6}
$$

These would be applied on one side of the shot and the opposite orientation would be applied on the other side. With many kinds of data sets, such as off-end marine recording in which a ship tows a hydrophone streamer, the above filter might be better in the interior too.

9.2.3. Are field arrays really needed?

Field arrays cancel random noise but their main function, I believe, is to cancel low-velocity coherent noises, something we now see is handled effectively by steep-dip deconvolution. While I do not advocate abandoning field arrays, it is pleasing to notice that with the arrival of steep-dip deconvolution, we are no longer so dependent on field arrays and perhaps coherent noises can be controlled where field arrays are impractical, as in certain 3-D geometries. A recently arrived 3-D shot profile from the sand dunes in the Middle East is Figure 9.11. The strong hyperbolas are ground roll seen in a line that does not include the shot. The open question here is, how should we formulate the problem of ground-roll removal in 3-D?

9.2.4. Which coefficients are really needed?

Steep-dip decon is a heavy consumer of computer time. Many small optimizations could be done, but more importantly, I feel there are some deeper issues that warrant further investigation. The first question is, how many filter coefficients should there be and where should they be? We would like to keep the number of nonzero filter coefficients to a minimum because it would speed the computation, but more

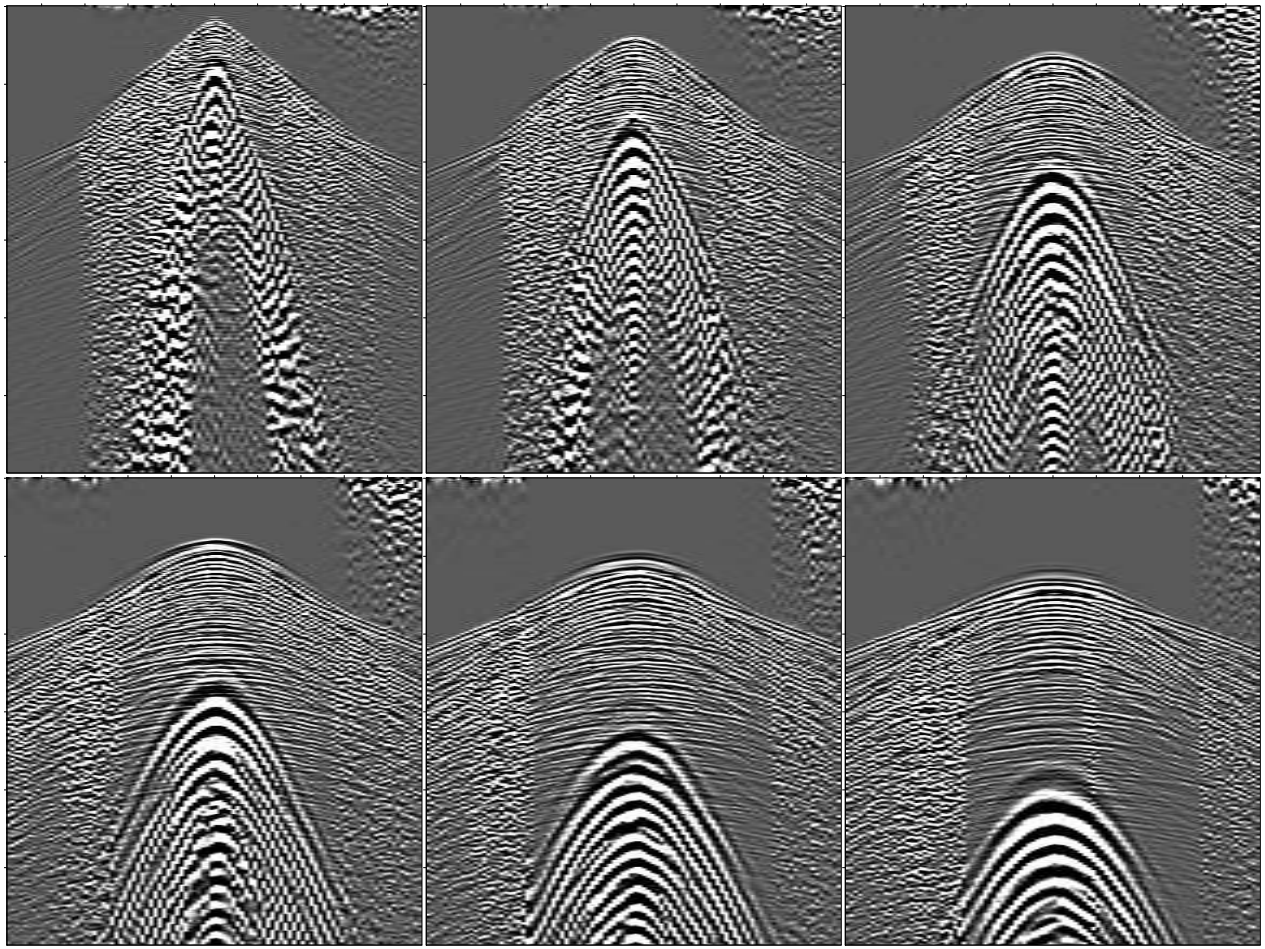


Figure 9.11: Sand dunes. One shot, six parallel receiver lines. [ER,M]

pch-dune3D

importantly I fear the filter output might be defective in some insidious way (perhaps missing primaries) when too many filter coefficients are used. Perhaps if 1-D decon were done sequentially with steep-dip decon the number of free parameters (and hence the amount of computer time) could be dropped even further. I looked at some of the filters and they scatter wildly with the Nyquist frequency (particularly those coefficients on the trace with the "1" constraint). This suggests using a damping term on the filter coefficients, after which perhaps the magnitude of a filter coefficient will be a better measure of whether this practice is really helpful. Also, it would, of course, be fun to get some complete data sets (rather than a single shot profile) to see the difference in the final stack.

9.3. INVERSION AND NOISE REMOVAL

Here we relate the basic theoretical statement of geophysical inverse theory to the basic theoretical statement of separation of signals from noises.

A common form of linearized geophysical inverse theory is

$$\mathbf{0} \quad \approx \quad \mathbf{W}(\mathbf{L}\mathbf{m} - \mathbf{d}) \tag{9.7}$$

$$\mathbf{0} \quad \approx \quad \epsilon \mathbf{A}\mathbf{m} \tag{9.8}$$

We choose the operator $\mathbf{L} = \mathbf{I}$ to be an identity and we rename the model $\mathbf{m}$ to be signal $\mathbf{s}$. Define noise by the decomposition of data into signal plus noise, so $\mathbf{n} = \mathbf{d} - \mathbf{s}$. Finally, let us rename the weighting (and filtering) operations $\mathbf{W} = \mathbf{N}$ on the noise and $\mathbf{A} = \mathbf{S}$ on the signal. Thus the usual model fitting becomes a fitting for signal-noise separation:

$$0 \quad \approx \quad \mathbf{N}(-\mathbf{n}) = \mathbf{N}(\mathbf{s} - \mathbf{d}) \tag{9.9}$$

$$0 \quad \approx \quad \epsilon \mathbf{S}\mathbf{s} \tag{9.10}$$

9.4. SIGNAL-NOISE DECOMPOSITION BY DIP

Choose noise $\mathbf{n}$ to be energy that has no spatial correlation and signal $\mathbf{s}$ to be energy with spatial correlation consistent with one, two, or possibly a few plane-wave segments. (Another view of noise is that a huge number of plane waves is required to

define the wavefield; in other words, with Fourier analysis you can make anything, signal or noise.) We know that a first-order differential equation can absorb (kill) a single plane wave, a second-order equation can absorb one or two plane waves, etc. In practice, we will choose the order of the wavefield and minimize power to absorb all we can, and call that the signal. **S** is the operator that absorbs (by prediction error) the plane waves and **N** absorbs noises and $\epsilon > 0$ is a small scalar to be chosen. The difference between **S** and **N** is the spatial order of the filters. Because we regard the noise as spatially uncorrelated, **N** has coefficients only on the time axis. Coefficients for **S** are distributed over time and space. They have one space level, plus another level for each plane-wave segment slope that we deem to be locally present. In the examples here the number of slopes is taken to be two. Where a data field seems to require more than two slopes, it usually means the "patch" could be made smaller.

It would be nice if we could forget about the goal (9.10) but without it the goal (9.9), would simply set the signal **s** equal to the data **d**. Choosing the value of ϵ will determine in some way the amount of data energy partitioned into each. The last thing we will do is choose the value of ϵ, and if we do not find a theory for it, we will experiment.

The operators **S** and **N** can be thought of as "leveling" operators. The method of least-squares sees mainly big things, and spectral zeros in **S** and **N** tend to cancel spectral lines and plane waves in **s** and **n**. (Here we assume that power levels remain fairly level in time. Were power levels to fluctuate in time, the operators **S** and **N** should be designed to level them out too.)

None of this is new or exciting in one dimension, but I find it exciting in more dimensions. In seismology, quasisinusoidal signals and noises are quite rare, whereas local plane waves are abundant. Just as a short one-dimensional filter can absorb a sinusoid of any frequency, a compact two-dimensional filter can absorb a wavefront of any dip.

To review basic concepts, suppose we are in the one-dimensional frequency domain. Then the solution to the fitting goals (9.10) and (9.9) amounts to minimizing a quadratic form by setting to zero its derivative, say

$$0 \quad = \quad \frac{\partial}{\partial \mathbf{s}'} \left((\mathbf{s}' - \mathbf{d}')\mathbf{N}'\mathbf{N}(\mathbf{s} - \mathbf{d}) + \epsilon^2 \mathbf{s}'\mathbf{S}'\mathbf{S}\mathbf{s} \right) \tag{9.11}$$

which gives the answer

$$\mathbf{s} \quad = \quad \left(\frac{\mathbf{N}'\mathbf{N}}{\mathbf{N}'\mathbf{N} + \epsilon^2 \mathbf{S}'\mathbf{S}} \right) \mathbf{d} \tag{9.12}$$

$$\mathbf{n} \quad = \quad \mathbf{d} - \mathbf{s} \quad = \quad \left(\frac{\epsilon^2 \mathbf{S}'\mathbf{S}}{\mathbf{N}'\mathbf{N} + \epsilon^2 \mathbf{S}'\mathbf{S}} \right) \mathbf{d} \tag{9.13}$$

To make this really concrete, consider its meaning in one dimension, where signal is white $\mathbf{S}'\mathbf{S} = 1$ and noise has the frequency ω_0, which is killable with the multiplier $\mathbf{N}'\mathbf{N} = (\omega - \omega_0)^2$. Now we recognize that equation (9.12) is a notch filter and equation (9.13) is a narrow-band filter.

The analytic solutions in equations (9.12) and (9.13) are valid in 2-D Fourier space or dip space too. I prefer to compute them in the time and space domain to give me tighter control on window boundaries, but the Fourier solutions give insight and offer a computational speed advantage.

Let us express the fitting goal in the form needed in computation.

$$\begin{bmatrix} \mathbf{0} \\ \mathbf{0} \end{bmatrix} \quad \approx \quad \begin{bmatrix} \mathbf{N} \\ \epsilon \mathbf{S} \end{bmatrix} \mathbf{s} + \begin{bmatrix} -\mathbf{N}\mathbf{d} \\ \mathbf{0} \end{bmatrix} \tag{9.14}$$

```
module signoi {
  use helicon
  use polydiv
  use solver_prc_mod
  use cgstep_mod
  integer, private                          :: i
  integer                                   :: niter, nd
  real                                      :: eps
  type( filter), dimension( :), pointer :: nn, ss
  real, dimension (nd), allocatable     :: dd
#% _init (nn, ss, niter, nd, eps)
i = 0
#% _lop (data, sign)
    integer stat1; i = i + 1
    call helicon_init (nn (i))
    call polydiv_init (nd, ss (i))
    stat1 = helicon_lop (.false., .false., data, dd)
    call solver_prc(m=sign, d=dd, Fop=helicon_lop, stepper=cgstep,
    niter=niter, Sop=polydiv_lop, nSop=nd, eps=eps)
    call cgstep_close()
#% _close
    call polydiv_close()
}
```

Back

signoi As with the missing-data subroutines, the potential number of iterations is large, because the dimensionality of the space of unknowns is much larger than the number of iterations we would find acceptable. Thus, sometimes changing the number of iterations `niter` can create a larger change than changing `epsilon`. Experience shows that helix preconditioning saves the day.

9.4.1. Signal/noise decomposition examples

Figure 9.12 demonstrates the signal/noise decomposition concept on synthetic data. The signal and noise have similar frequency spectra but different dip spectra.

Before I discovered helix preconditioning, Ray Abma found that different results were obtained when the fitting goal was cast in terms of **n** instead of **s**. Theoretically it should not make any difference. Now I believe that with preconditioning, or even without it, if there are enough iterations, the solution should be independent of whether the fitting goal is cast with either **n** or **s**.

Figure 9.13 shows the result of experimenting with the choice of ϵ. As expected, increasing ϵ weakens **s** and increases **n**. When ϵ is too small, the noise is small and the signal is almost the original data. When ϵ is too large, the signal

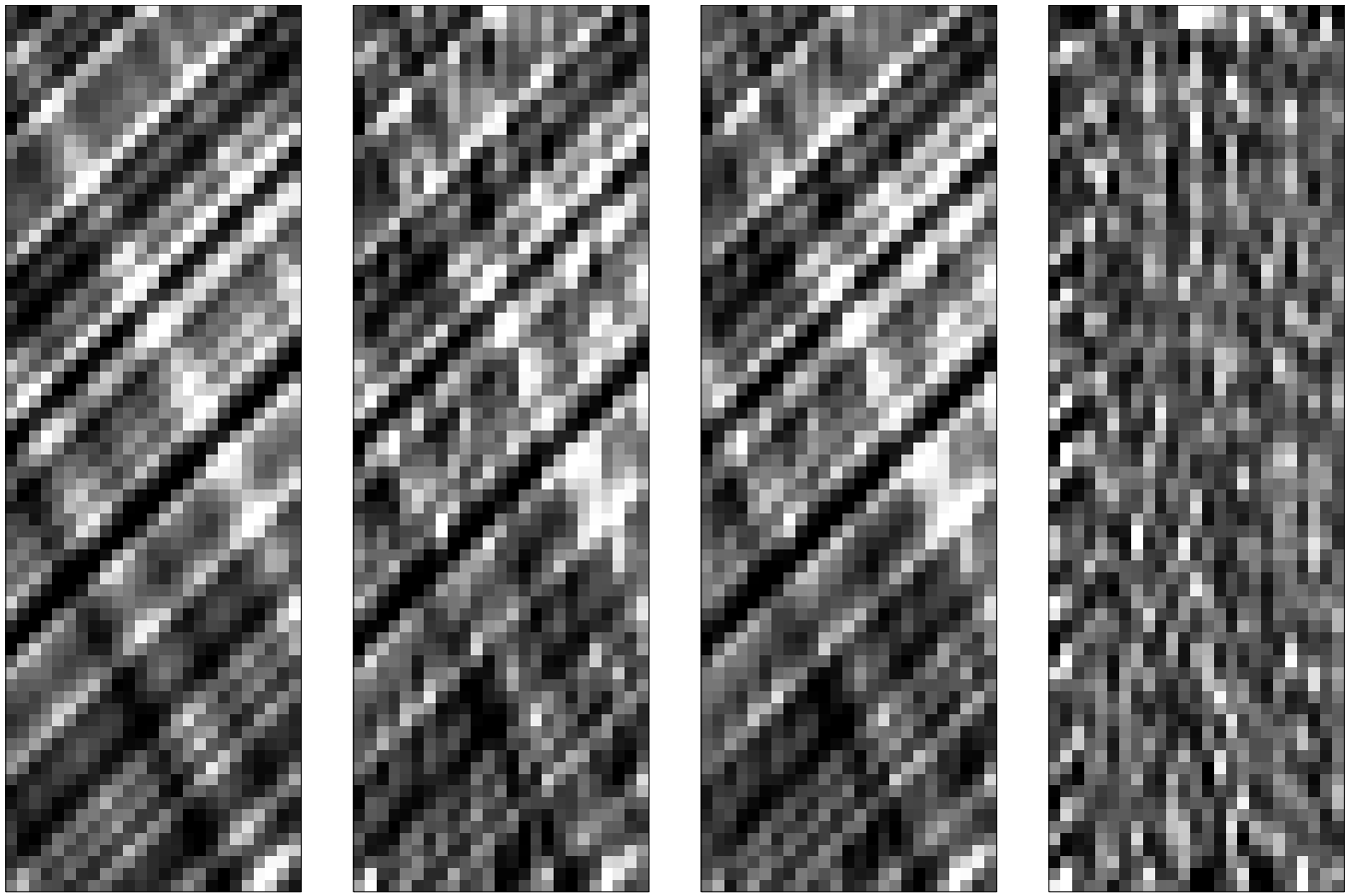


Figure 9.12: The input signal is on the left. Next is that signal with noise added. Next, for my favorite value of `epsilon=1.`, is the estimated signal and the estimated noise. pch-signoi90 [ER]

is small and coherent events are pushed into the noise. (Figure 9.13 rescales both signal and noise images for the clearest display.)

Notice that the leveling operators **S** and **N** were both estimated from the original signal and noise mixture $\mathbf{d} = \mathbf{s} + \mathbf{n}$ shown in Figure 9.12. Presumably we could do even better if we were to reestimate **S** and **N** from the estimates **s** and **n** in Figure 9.13.

9.4.2. Spitz for variable covariances

Since signal and noise are uncorrelated, the spectrum of data is the spectrum of the signal plus that of the noise. An equation for this idea is

$$\sigma_d^2 \quad = \quad \sigma_s^2 \; + \; \sigma_n^2 \tag{9.15}$$

This says resonances in the signal and resonances in the noise will both be found in the data. When we are given σ_d^2 and σ_n^2 it seems a simple matter to subtract to get σ_s^2. Actually it can be very tricky. We are never given σ_d^2 and σ_n^2; we must estimate them. Further, they can be a function of frequency, wave number, or dip, and these can be changing during measurements. We could easily find ourselves

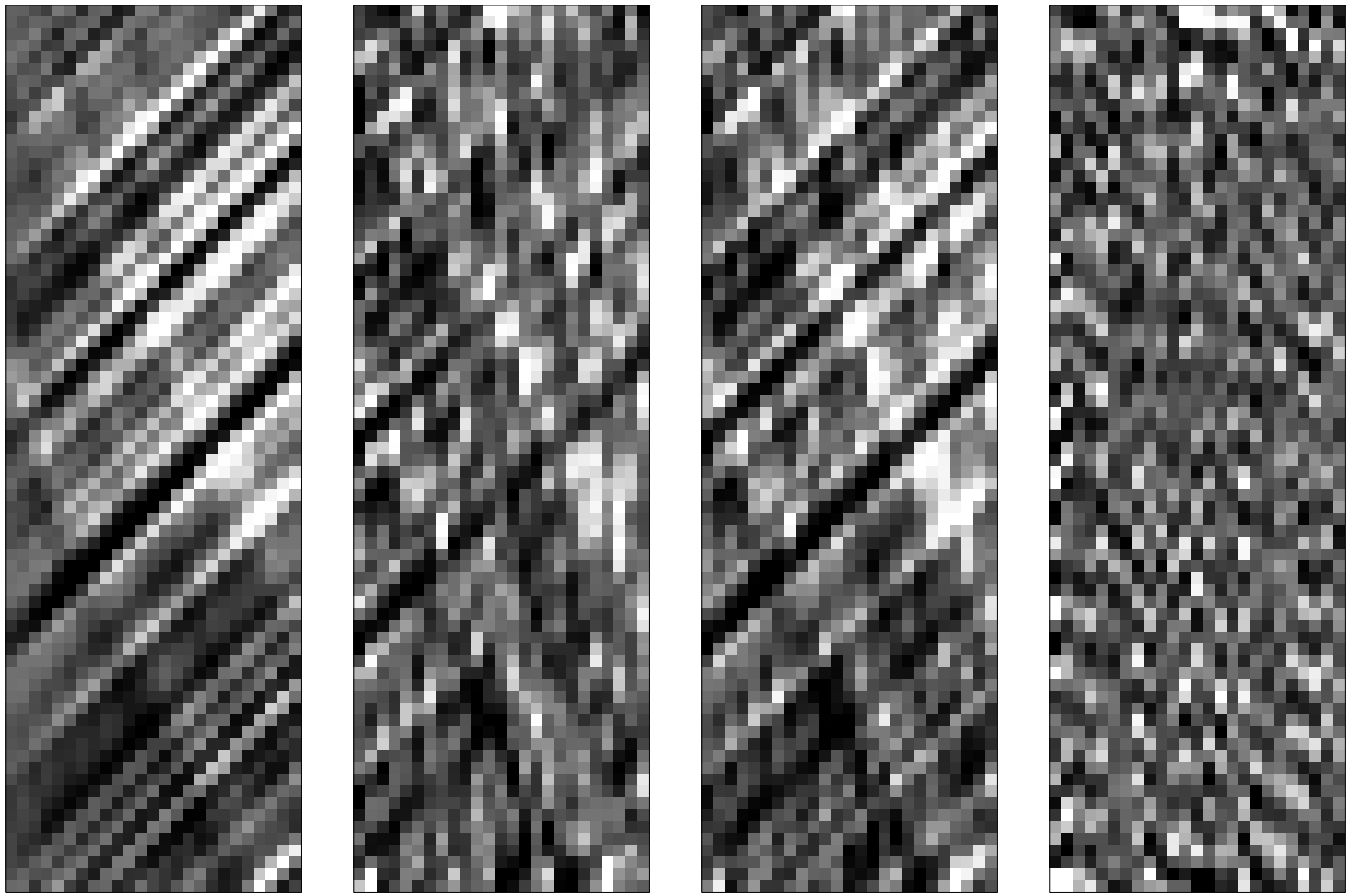


Figure 9.13: Left is an estimated signal-noise pair where `epsilon=4` has improved the appearance of the estimated signal but some coherent events have been pushed into the noise. Right is a signal-noise pair where `epsilon=.25`, has improved the appearance of the estimated noise but the estimated signal looks no better than original data. pch-signeps90 [ER]

with a negative estimate for σ_s^2 which would ruin any attempt to segregate signal from noise. An idea of Simon Spitz can help here.

Let us reexpress equation (9.15) with prediction-error filters.

$$\frac{1}{\bar{A}_d A_d} \quad = \quad \frac{1}{\bar{A}_s A_s} + \frac{1}{\bar{A}_n A_n} \quad = \quad \frac{\bar{A}_s A_s \; + \; \bar{A}_n A_n}{(\bar{A}_s A_s)(\bar{A}_n A_n)} \tag{9.16}$$

Inverting

$$\bar{A}_d A_d \quad = \quad \frac{(\bar{A}_s A_s)\,(\bar{A}_n A_n)}{\bar{A}_s A_s \; + \; \bar{A}_n A_n} \tag{9.17}$$

The essential feature of a PEF is its zeros. Where a PEF approaches zero, its inverse is large and resonating. When we are concerned with the zeros of a mathematical function we tend to focus on numerators and ignore denominators. The zeros in $\bar{A}_s A_s$ compound with the zeros in $\bar{A}_n A_n$ to make the zeros in $\bar{A}_d A_d$. This motivates the "Spitz Approximation."

$$\bar{A}_d A_d \quad = \quad (\bar{A}_s A_s)\,(\bar{A}_n A_n) \tag{9.18}$$

It usually happens that we can find a patch of data where no signal is present. That's a good place to estimate the noise PEF A_n. It is usually much harder to find

a patch of data where no noise is present. This motivates the Spitz approximation which by saying $A_d = A_s A_n$ tells us that the hard-to-estimate A_s is the ratio $A_s = A_d / A_n$ of two easy-to-estimate PEFs.

It would be computationally convenient if we had A_s expressed not as a ratio. For this, form the signal $\mathbf{u} = \mathbf{A}_n \mathbf{d}$ by applying the noise PEF A_n to the data $\mathbf{d}$. The spectral relation is

$$\sigma_u^2 \quad = \quad \sigma_d^2 / \sigma_n^2 \tag{9.19}$$

Inverting this expression and using the Spitz approximation we see that a PEF estimate on $\mathbf{u}$ is the required A_s in numerator form because

$$A_u \quad = \quad A_d / A_n \quad = \quad A_s \tag{9.20}$$

9.4.3. Noise removal on Shearer's data

Professor Peter Shearer[1] gathered the earthquakes from the IDA network, an array of about 25 widely distributed gravimeters, donated by Cecil Green, and Shearer

[1] I received the data for this stack from Peter Shearer at the Cecil and Ida Green Institute of Geophysics and Planetary Physics of the Scripps Oceanographic In-

selected most of the shallow-depth earthquakes of magnitude greater than about 6 over the 1981-91 time interval, and sorted them by epicentral distance into bins 1° wide and stacked them. He generously shared his edited data with me and I have been restacking it, compensating for amplitude in various ways, and planning time and filtering compensations.

Figure 9.14 shows a test of noise subtraction by multidip narrow-pass filtering on the Shearer-IDA stack. As with prediction there is a general reduction of the noise. Unlike with prediction, weak events are preserved and noise is subtracted from them too.

Besides the difference in theory, the separation filters are much smaller because their size is determined by the concept that "two dips will fit anything locally" (`a2=3`), versus the prediction filters "needing a sizeable window to do statistical av-

---

stitute. I also received his permission to redistribute it to friends and colleagues. Should you have occasion to copy it please reference (Shearer, 1991a) (Shearer, 1991b) it properly. Examples of earlier versions of these stacks are found in the references. Professor Shearer may be willing to supply newer and better stacks. His electronic mail address is `shearer@mahi.ucsd.edu`.

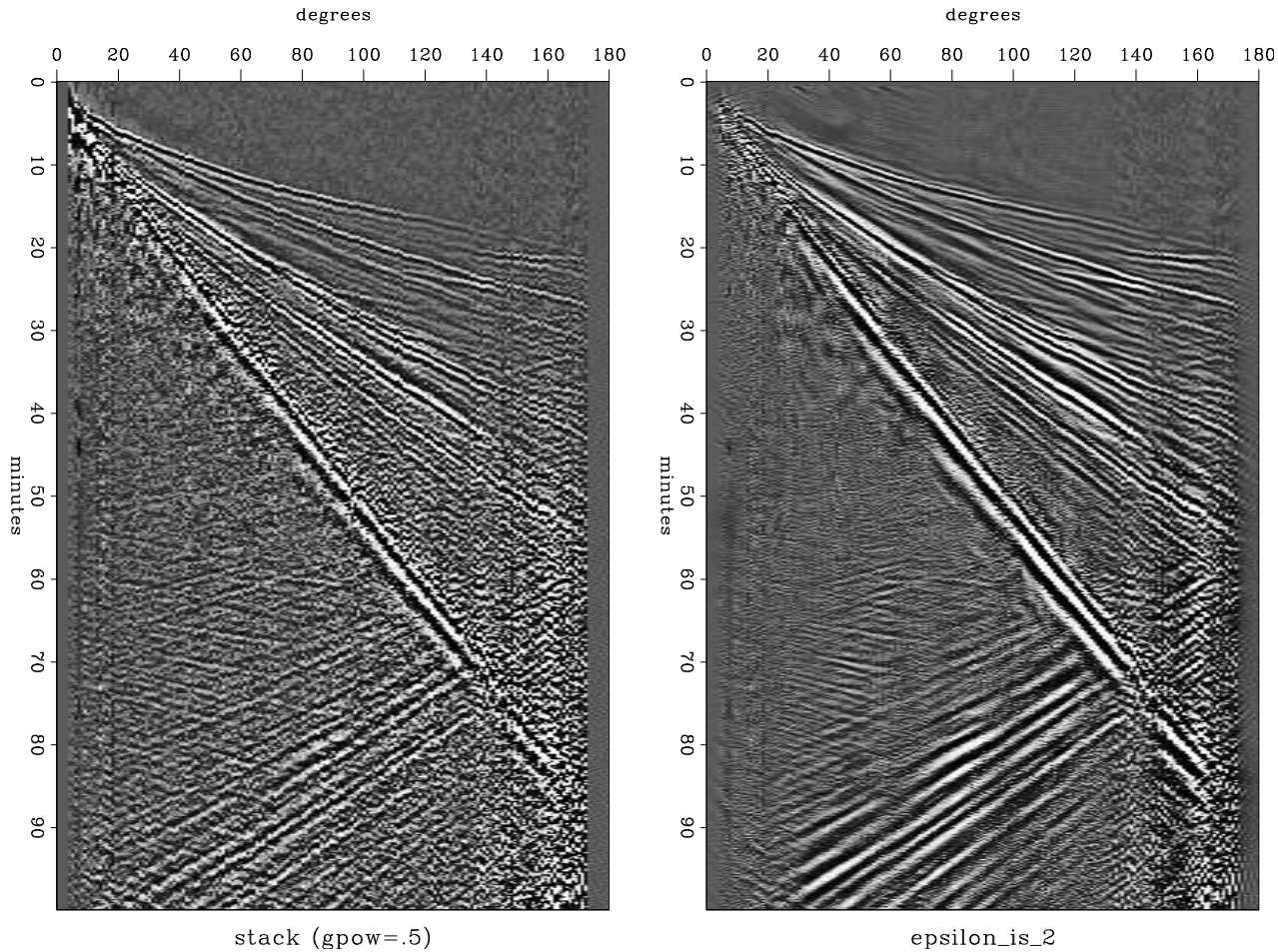


Figure 9.14: Stack of Shearer's IDA data (left). Multidip filtered (right). It is pleasing that the noise is reduced while weak events are preserved. pch-sneps+290 [CR,M]

eraging." The same aspect ratio `a1/a2` is kept and the page is now divided into 11 vertical patches and 24 horizontal patches (whereas previously the page was divided in 3×4 patches). In both cases the patches overlap about 50%. In both cases I chose to have about ten times as many equations as unknowns on each axis in the estimation. The ten degrees of freedom could be distributed differently along the two axes, but I saw no reason to do so.

9.4.4. The human eye as a dip filter

Although the filter seems to be performing as anticipated, no new events are apparent. I believe the reason that we see no new events is that the competition is too tough. We are competing with the human eye, which through aeons of survival has become is a highly skilled filter. Does this mean that there is no need for filter theory and filter subroutines because the eye can do it equally well? It would seem so. Why then pursue the subject matter of this book?

The answer is 3-D. The human eye is not a perfect filter. It has a limited (though impressive) dynamic range. A nonlinear display (such as wiggle traces) can prevent it from averaging. The eye is particularly good at dip filtering, because the paper

can be looked at from a range of grazing angles and averaging window sizes miraculously adjust to the circumstances. The eye can be overwhelmed by too much data. The real problem with the human eye is that the retina is only two-dimensional. The world contains many three-dimensional data volumes. I don't mean the simple kind of 3-D where the contents of the room are nicely mapped onto your 2-D retina. I mean the kind of 3-D found inside a bowl of soup or inside a rock. A rock can be sliced and sliced and sliced again and each slice is a picture. The totality of these slices is a movie. The eye has a limited ability to deal with movies by optical persistence, an averaging of all pictures shown in about 1/10 second interval. Further, the eye can follow a moving object and perform the same averaging. I have learned, however, that the eye really cannot follow two objects at two different speeds and average them both over time. Now think of the third dimension in Figure 9.14. It is the dimension that I summed over to make the figure. It is the 1° range bin. If we were viewing the many earthquakes in each bin, we would no longer be able to see the out-of-plane information which is the in-plane information in Figure 9.14.

To view genuinely 3-D information we must see a movie, or we must compress the 3-D to 2-D. There are only a small number of ways to compress 3-D to 2-D. One is to select planes from the volume. One is to sum the volume over one

of its axes, and the other is a compromise, a filtering over the axis we wish to abandon before subsampling on it. That filtering is a local smoothing. If the local smoothing has motion (out of plane dip) of various velocities (various dips), then the desired process of smoothing the out of plane direction is what we did in the in-plane direction in Figure 9.14. But Figure 9.14 amounts to more than that. It amounts to a kind of simultaneous smoothing in the *two* most coherent directions whereas in 3-D your eye can smooth in only *one* direction when you turn your head along with the motion.

> If the purpose of data processing is to collapse 3-D data volumes to 2-D where they are comprehensible to the human eye, then perhaps data-slope adaptive, low-pass filtering in the out-of-plane direction is the best process we can invent.

My purpose in filtering the earthquake stacks is to form a guiding "pilot trace" to the analysis of the traces *within* the bin. Within each bin, each trace needs small time shifts and perhaps a small temporal filter to best compensate it to . . . to what? to the pilot trace, which in these figures was simply a stack of traces in the bin. Now that we have filtered in the range direction, however, the next stack can be made with a better quality pilot.

9.5. SPACE-VARIABLE DECONVOLUTION

Filters sometimes change with time and space. We sometimes observe signals whose spectrum changes with position. A filter that changes with position is called nonstationary. We need an extension of our usual convolution operator `hconest` /prog:hconest. Conceptually, little needs to be changed besides changing `aa(ia)` to `aa(ia,iy)`. But there is a practical problem. Fomel and I have made the decision to clutter up the code somewhat to save a great deal of memory. This should be important to people interested in solving multidimensional problems with big data sets.

Normally, the number of filter coefficients is many fewer than the number of data points, but here we have very many more. Indeed, there are `na` times more. Variable filters require `na` times more memory than the data itself. To make the nonstationary helix code more practical, we now require the filters to be constant in patches. The data type for nonstationary filters (which are constant in patches) is introduced in module `nhelix`, which is a simple modification of module `helix` /prog:helix. nhelix What is new is the integer valued vector `pch(nd)` the size of the one-dimensional (helix) output data space. Every filter output point is to be assigned to a patch. All filters of a given patch number will be the same filter.

```
module nhelix {                                  # Define nonstationary filter type
  use helix
  type nfilter {                                          #  nd  is data length.
    logical,       dimension(:), pointer :: mis  # (nd) boundary conditions
    integer,       dimension(:), pointer :: pch  # (nd) patches
    type( filter), dimension(:), pointer :: hlx  # (np) filters
  }
contains
  subroutine nallocate( aa, nh, pch) {          # allocate a filter
    type( nfilter)                             :: aa
    integer, dimension(:), intent( in)   :: nh, pch
    integer                                    :: ip, np, nd
    np = size( nh); allocate( aa%hlx( np))
    do ip = 1, np
       call allocatehelix( aa%hlx( ip), nh( ip))
    nd = size( pch); allocate( aa%pch( nd))
    aa%pch = pch
    nullify( aa%mis)                                    # set null pointer for mis.
  }
  subroutine ndeallocate( aa) {                         # destroy a filter
    type( nfilter) :: aa
    integer        :: ip
    do ip = 1, size( aa%hlx)
            call deallocatehelix( aa%hlx( ip))
    deallocate( aa%hlx, aa%pch)
    if( associated( aa%mis))                            # if logicals were allocated
        deallocate( aa%mis)                             # free them
  }
}
```

Back

```
module createnhelixmod {      # Create non-stationary helix filter lags and mis
use createhelixmod
use nhelix
use nbound
contains
  function createnhelix( nd, center, gap, na, pch) result (nsaa) {
    type( nfilter)                        :: nsaa   # needed by nhelicon
    integer, dimension(:), intent(in)  :: nd, na # data and filter axes
    integer, dimension(:), intent(in)  :: center # normally (na1/2,na2/2,...,1)
    integer, dimension(:), intent(in)  :: gap    # normally ( 0,   0,  0,...,0)
    integer, dimension(:), intent(in)  :: pch    # (prod(nd)) patching
    type( filter)                         :: aa
    integer                               :: n123, np, ip
    integer, dimension(:), allocatable :: nh
    aa = createhelix( nd, center, gap, na)
    n123 = product( nd);  np = maxval(pch)
    allocate (nh (np)); nh = size (aa%lag)
    call nallocate( nsaa, nh, pch)
    do ip = 1, np
       nsaa%hlx( ip)%lag = aa%lag
    call deallocatehelix (aa)
    call nboundn(1, nd, na, nsaa)
    }
}
```

Back

Nonstationary helixes are created with `createnhelixmod`, which is a simple modification of module `createhelixmod` /prog:createhelixmod. createnhelixmod
Notice that the user must define the `pch(product(nd))` vector before creating a nonstationary helix. For a simple 1-D time-variable filter, presumably `pch` would be something like $(1,1,2,2,3,3,\cdots)$. For multidimensional patching we need to think a little more.

Finally, we are ready for the convolution operator. The operator `nhconest` /prog:nhconest allows for a different filter in each patch. nhconest A filter output `y(iy)` has its filter from the patch `ip=aa%pch(iy)`. The line `t=a(ip,:)` extracts the filter for the `ip`th patch. If you are confused (as I am) about the difference between `aa` and `a`, maybe now is the time to have a look at beyond Loptran to the Fortran version.[2]

Because of the massive increase in the number of filter coefficients, allowing these many filters takes us from overdetermined to very undetermined. We can estimate all these filter coefficients by the usual deconvolution fitting goal (6.18)

$$\mathbf{0} \quad \approx \quad \mathbf{r} \quad = \quad \mathbf{YKa} + \mathbf{r}_0 \tag{9.21}$$

[2] http://sepwww.stanford.edu/sep/prof/gee/Lib/

```
module nhconest {           # Nonstationary convolution, adjoint is the filter.
use nhelix
  real, dimension(:), pointer                :: x
  type( nfilter)                             :: aa
  integer                                    :: nhmax
  integer, private                           :: np
#% _init( x, aa, nhmax)
    np = size( aa%hlx)
#% _lop( a( nhmax, np), y(:))
    integer                             :: ia, ix, iy, ip
    integer, dimension(:), pointer :: lag
    do iy = 1, size( y) {  if( aa%mis( iy)) cycle
        ip = aa%pch( iy); lag => aa%hlx( ip)%lag
        do ia = 1, size( lag) {
                ix = iy - lag( ia);   if( ix < 1) cycle
                    if( adj)    a( ia, ip) +=  y( iy) * x( ix)
                    else        y( iy) +=  a( ia, ip) * x( ix)
                 }
        }
}
```

Back

but we need to supplement it with some damping goals, say

$$\begin{aligned} \mathbf{0} &\approx \mathbf{YKa} + \mathbf{r}_0 \\ \mathbf{0} &\approx \epsilon\, \mathbf{Ra} \end{aligned} \tag{9.22}$$

where $\mathbf{R}$ is a roughening operator to be chosen.

Experience with missing data in Chapter 3 shows that when the roughening operator $\mathbf{R}$ is a differential operator, the number of iterations can be large. We can speed the calculation immensely by "preconditioning". Define a new variable $\mathbf{m}$ by $\mathbf{a} = \mathbf{R}^{-1}\mathbf{m}$ and insert it into (9.22) to get the equivalent preconditioned system of goals.

$$\mathbf{0} \approx \mathbf{YKR}^{-1}\mathbf{m} \tag{9.23}$$

$$\mathbf{0} \approx \epsilon\, \mathbf{m} \tag{9.24}$$

The fitting (9.23) uses the operator $\mathbf{YKR}^{-1}$. For $\mathbf{Y}$ we can use subroutine `nhconest()` /prog:nhconest; for the smoothing operator $\mathbf{R}^{-1}$ we can use nonstationary polynomial division with operator `npolydiv()`: npolydiv Now we have all the pieces we need. As we previously estimated stationary filters with the module `pef` /prog:pef, now we can estimate nonstationary PEFs with the module `npef`

```
module npolydiv {                              # Helix polynomial division
use nhelix
integer                             :: nd
type( nfilter)                      :: aa
real, dimension (nd), allocatable :: tt
#%  _init ( nd, aa)
#%  _lop  ( xx, yy)
integer                           :: ia, ix, iy, ip
integer, dimension(:), pointer :: lag
real,    dimension(:), pointer :: flt
tt = 0.
if( adj) {
        tt = yy
        do iy= nd, 1, -1 { ip = aa%pch( iy)
                lag => aa%hlx( ip)%lag; flt => aa%hlx( ip)%flt
                do ia = 1, size( lag) {
                        ix = iy - lag( ia);     if( ix < 1)  cycle
                        tt( ix) -=  flt( ia) * tt( iy)
                        }
                }
        xx += tt
 } else {
        tt = xx
        do iy= 1, nd { ip = aa%pch( iy)
                lag => aa%hlx( ip)%lag; flt => aa%hlx( ip)%flt
                do ia = 1, size( lag) {
                        ix = iy - lag( ia);      if( ix < 1)  cycle
                        tt( iy) -=  flt( ia) * tt( ix)
                        }
                }
        yy += tt
        }
}
```

Back

```
module npef {                                   # Estimate non-stationary PEF
  use nhconest
  use npolydiv2
  use cgstep_mod
  use solver_mod
contains
  subroutine find_pef( dd, aa, rr, niter, eps, nh) {
    integer,          intent( in)    :: niter, nh      # number of iterations
    real,             intent( in)    :: eps            # epsilon
    type( nfilter)                   :: aa             # estimated PEF output.
    type( nfilter),   intent( in)    :: rr             # roughening filter.
    real,    dimension(:), pointer   :: dd             # input data
    integer                          :: ip, ih, np, nr # filter lengths
    real, dimension (:), allocatable :: flt            # np*na filter coefs
    np = size( aa%hlx)                  # data   length
    nr = np*nh
    allocate( flt( nr))
    call nhconest_init( dd, aa, nh)
    call npolydiv2_init( nr, rr)
    call solver_prec( nhconest_lop, cgstep, x= flt, dat= -dd, niter= niter,
                      prec= npolydiv2_lop, nprec= nr, eps= eps)
    call   cgstep_close()
    call npolydiv2_close()
    call nhconest_close()
    do ip = 1, np
       do ih = 1, size( aa%hlx(ip)%lag)
          aa%hlx( ip)%flt( ih) = flt( (ip-1)*nh + ih)
    deallocate( flt)
    }
}
```

Back

/prog:npef. The steps are hardly any different. npef Near the end of module `npef` is a filter `reshape` from a 1-D array to a 2-D array. If you find it troublesome that `nhconest` /prog:nhconest was using the filter during the optimization as already multidimensional, perhaps again, it is time to examine the Fortran code. The answer is that there has been a conversion back and forth partially hidden by Loptran.

Figure 9.15 shows a synthetic data example using these programs. As we hope for deconvolution, events are compressed. The compression is fairly good, even though each event has a different spectrum. What is especially pleasing is that satisfactory results are obtained in truly small numbers of iterations (about three). The example is for two free filter coefficients $(1, a_1, a_2)$ per output point. The roughening operator $\mathbf{R}$ was taken to be $(1, -2, 1)$ which was factored into causal and anticausal finite difference.

I hope also to find a test case with field data, but experience in seismology is that spectral changes are slow, which implies unexciting results. Many interesting examples should exist in two- and three-dimensional filtering, however, because reflector dip is always changing and that changes the *spatial* spectrum.

In multidimensional space, the smoothing filter $\mathbf{R}^{-1}$ can be chosen with inter-

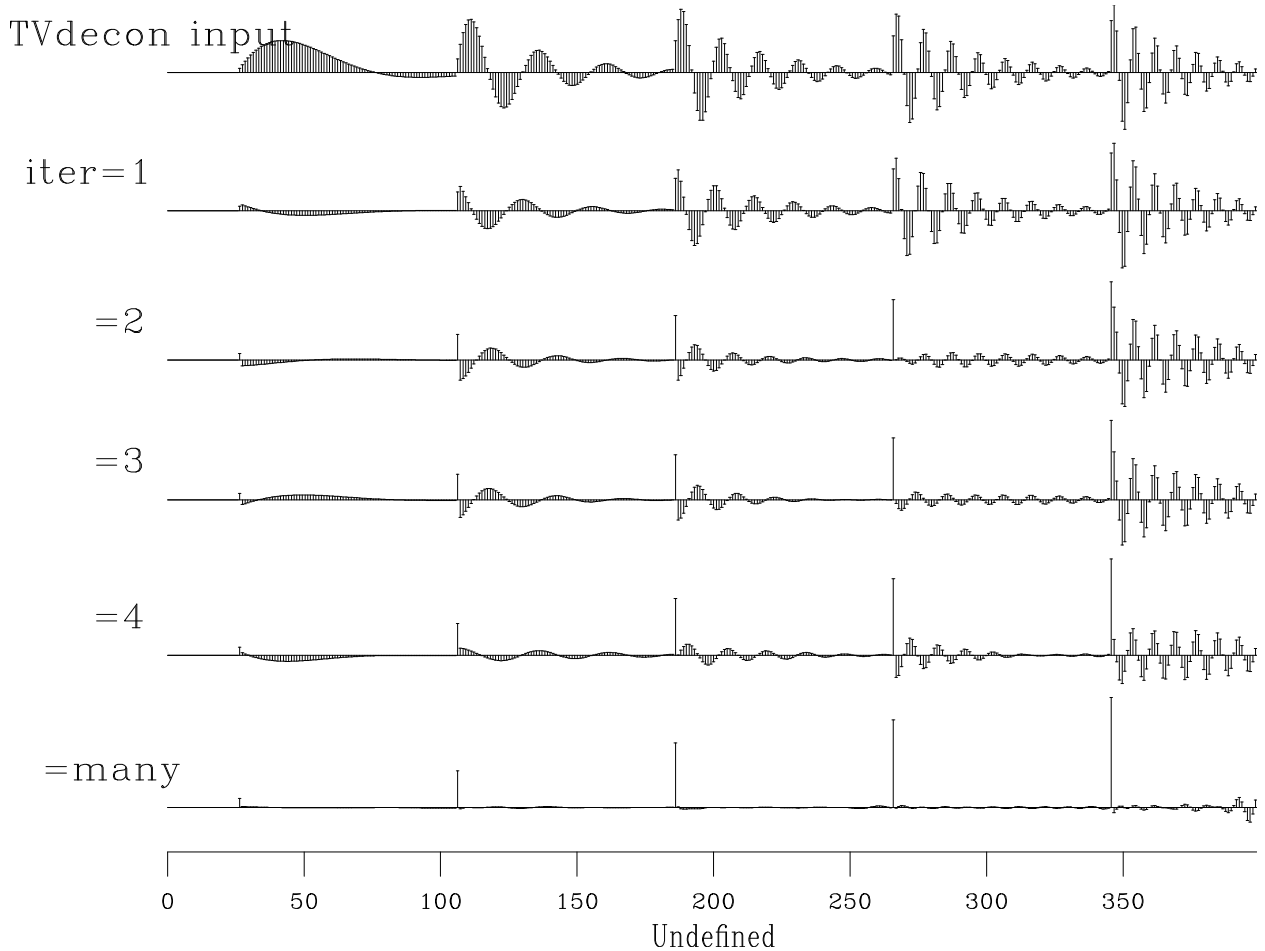


Figure 9.15: Time variable deconvolution with two free filter coefficients and a gap of 6. pch-tvdecon90 [ER]

esting directional properties. Sergey, Bob, Sean and I have joked about this code being the "double helix" program because there are two multidimensional helixes in it, one the smoothing filter, the other the deconvolution filter. Unlike the biological helixes, however, these two helixes do not seem to form a symmetrical pair.

EXERCISES:

1 Is `nhconest` /prog:nhconest the inverse operator to `npolydiv` /prog:npolydiv? Do they commute?

2 Sketch the matrix corresponding to operator `nhconest` /prog:nhconest. HINTS: Do not try to write all the matrix elements. Instead draw short lines to indicate rows or columns. As a "warm up" consider a simpler case where one filter is used on the first half of the data and another filter for the other half. Then upgrade that solution from two to about ten filters.

REFERENCES

Shearer, P. M., 1991a, Constraints on upper mantle discontinuities from observations of long period reflected and converted phases: J. Geophys. Res., **96**, no.

B11, 18147–18182. 643

Shearer, P. M., 1991b, Imaging global body wave phases by stacking long-period seismograms: J. Geophys. Res., **96**, no. B12, 20535–20324. 643

Chapter 10

Plane waves in three dimensions

In this chapter we seek a deeper understanding of plane waves in *three* dimensions, where the examples and theory typically refer to functions of time t and two space coordinates (x, y), or to 3-D migration images where the t coordinate is depth or

traveltime depth. As in Chapter 9, we need to decompose data volumes into sub-cubes, shown in Figure 10.1.

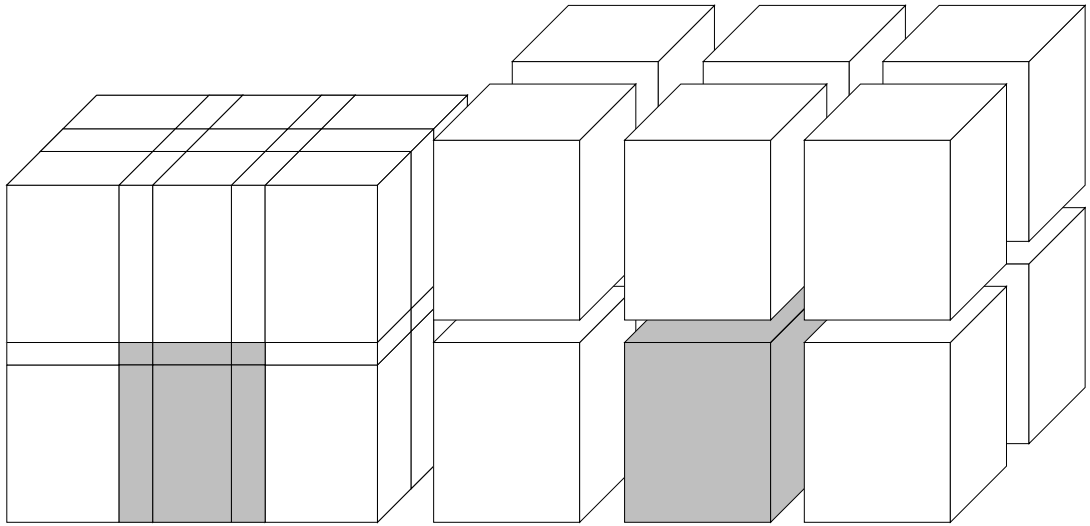


Figure 10.1: Left is space of inputs and outputs. Right is their separation during analysis. lmn-rayab3D [NR]

In this chapter we will see that the wave model implies the 3-D whitener is not a cube filter but two planar filters. The wave model allows us to determine the scale factor of a signal, even where signals fluctuate in strength because of interference. Finally, we examine the local-monoplane concept that uses the superposition principle to distinguish a sedimentary model cube from a data cube.

10.1. THE LEVELER: A VOLUME OR TWO PLAN

In *two* dimensions, levelers were taken to be PEFs, small rectangular planes of numbers in which the time axis included enough points to include reasonable stepouts were included and the space axis contained one level plus another space level, for each plane-wave slope supposed to be present.

We saw that a whitening filter in *three* dimensions is a small volume with shape defined by subroutine `createhelix()`. It might seem natural that the number of points on the x- and y-axes be related to the number of plane waves present. Instead, I assert that if the volume contains plane waves, we don't want a *volume* filter to whiten it; we can use a *pair* of *planar* filters to do so and the order of those filters is the number of planes thought to be simultaneously present. I have no firm mathematical proofs, but I offer you some interesting discussions, examples, and computer tools for you to experiment with. It seems that some applications call for the volume filter while others call for the two planes. Because two planes of numbers generally contain many fewer adjustable values than a volume, statistical-estimation reasons also favor the planes.

What is the lowest-order filter that, when applied to a volume, will destroy one and only one slope of plane wave?

First we seek the answer to the question, "What is the lowest order filter that will destroy one and only one plane?" To begin with, we consider that plane to be horizontal so the volume of numbers is $f(t,x,y) = b(t)$ where $b(t)$ is an arbitrary function of time. One filter that has zero-valued output (destroys the plane) is $\partial_x \equiv \partial/\partial x$. Another is the operator $\partial_y \equiv \partial/\partial y$. Still another is the Laplacian operator which is $\partial_{xx} + \partial_{yy} \equiv \partial^2/\partial x^2 + \partial^2/\partial y^2$.

The problem with $\partial/\partial x$ is that although it destroys the required plane, it also destroys $f(t,x,y) = a(t,y)$ where $a(t,y)$ is an *arbitrary* function of (t,y) such as a cylinder with axis parallel to the x-axis. The operator $\partial/\partial y$ has the same problem but with the axes rotated. The Laplacian operator not only destroys our desired plane, but it also destroys the well known nonplanar function $e^{ax}\cos(ay)$, which is just one example of the many other interesting shapes that constitute solutions to Laplace's equation.

I remind you of a basic fact: When we set up the fitting goal $\mathbf{0} \approx \mathbf{Af}$, the quadratic form minimized is $\mathbf{f}'\mathbf{A}'\mathbf{Af}$, which by differentiation with respect to $\mathbf{f}'$ gives

us (in a constraint-free region) $\mathbf{A}'\mathbf{A}\mathbf{f} = \mathbf{0}$. So, minimizing the volume integral (actually the sum) of the squares of the components of the gradient implies that Laplace's equation is satisfied.

In any volume, the lowest-order filter that will destroy level planes and no other wave slope is a filter that has one input and *two outputs*. That filter is the gradient, $(\partial/\partial x, \partial/\partial y)$. Both outputs vanish if and only if the plane has the proper horizontal orientation. Other objects and functions are not extinguished (except for the non-wave-like function $f(t,x,y) = \text{const}$). It is annoying that we must deal with *two* outputs and that will be the topic of further discussion.

A wavefield of tilted parallel planes is $f(t,x,y) = g(\tau - p_x x - p_y y)$, where $g()$ is an arbitrary one-dimensional function. The operator that destroys these tilted planes is the two-component operator $(\partial_x + p_x \partial_t,\ \partial_y + p_y \partial_t)$.

The operator that destroys a family of dipping planes

$$f(t,x,y) \quad = \quad g(\tau - p_x x - p_y y)$$

is

$$\begin{bmatrix} \frac{\partial}{\partial x} + p_x \frac{\partial}{\partial t} \\ \\ \frac{\partial}{\partial y} + p_y \frac{\partial}{\partial t} \end{bmatrix}$$

10.1.1. PEFs overcome spatial aliasing of difference operators

The problem I found with finite-difference representations of differential operators is that they are susceptible to spatial aliasing. Even before they encounter spatial aliasing, they are susceptible to accuracy problems known in finite-difference wave propagation as "frequency dispersion." The aliasing problem can be avoided by the

use of spatial prediction operators such as

$$\begin{matrix} \cdot & a \\ \cdot & b \\ 1 & c \\ \cdot & d \\ \cdot & e \end{matrix} \tag{10.1}$$

where the vertical axis is time; the horizontal axis is space; and the "·"s are zeros. Another possibility is the 2-D whitening filter

$$\begin{matrix} f & a \\ g & b \\ 1 & c \\ \cdot & d \\ \cdot & e \end{matrix} \tag{10.2}$$

Imagine all the coefficients vanished but $d = -1$ and the given 1. Such filters would annihilate the appropriately sloping plane wave. Slopes that are not exact integers are also approximately extinguishable, because the adjustable filter coefficients can interpolate in time. Filters like (10.2) do the operation $\partial_x + p_x \partial_t$, which is a com-

ponent of the gradient in the plane of the wavefront, *and* they include a temporal deconvolution aspect and a spatial coherency aspect. My experience shows that the operators (10.1) and (10.2) behave significantly differently in practice, and I am not prepared to fully explain the difference, but it seems to be similar to the gapping of one-dimensional filters.

You might find it alarming that your teacher is not fully prepared to explain the difference between a volume and two planes, but please remember that we are talking about the factorization of the volumetric spectrum. Spectral matrices are well known to have structure, but books on theory typically handle them as simply $\lambda\mathbf{I}$. Anyway, wherever you see an $\mathbf{A}$ in a three-dimensional context, you may wonder whether it should be interpreted as a cubic filter that takes one volume to another, or as two planar filters that take one volume to two volumes as shown in Figure 10.2.

10.1.2. My view of the world

I start from the idea that the four-dimensional world (t, x, y, z) is filled with expanding spherical waves and with quasispherical waves that result from reflection from quasiplanar objects and refraction through quasihomogeneous materials. We rarely,

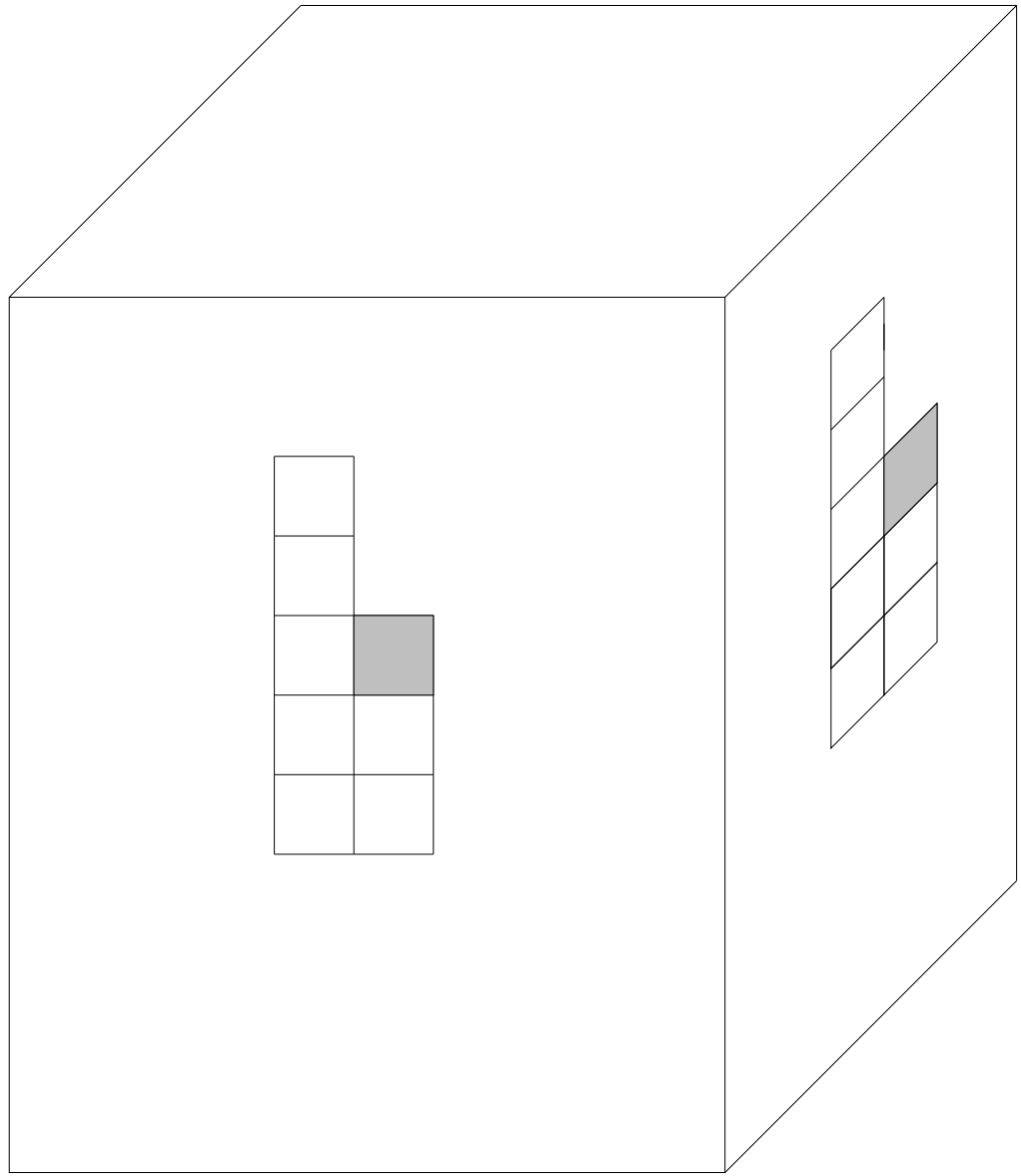


Figure 10.2: An inline 2-D PEF and a crossline 2-D PEF both applied throughout the volume. To find each filter, minimize each output power independently. lmn-rayab3Doper [NR]

if ever see in an observational data cube, an entire expanding spherical wave, but we normally have a two- or three-dimensional slice with some wavefront curvature. We analyze data subcubes that I call bricks. In any brick we see only local patches of apparent plane waves. I call them platelets. From the microview of this brick, the platelets come from the "great random-point-generator in the sky," which then somehow convolves the random points with a platelike impulse response. If we could deconvolve these platelets back to their random source points, there would be nothing left inside the brick because the energy would have gone outside. We would have destroyed the energy inside the brick. If the platelets were coin shaped, then the gradient magnitude would convert each coin to its circular rim. The plate sizes and shapes are all different and they damp with distance from their centers, as do Gaussian beams. If we observed *rays* instead of wavefront platelets then we might think of the world as being filled with noodles, and then. . . .

How is it possible that in a small brick we can do something realistic about deconvolving a spheroidal impulse response that is much bigger than the brick? The same way as in one dimension, where in a small time interval we can estimate the correct deconvolution filter of a long resonant signal. A three-point filter destroys a sinusoid.

The inverse filter to the expanding spherical wave might be a huge cube. Good approximations to this inverse at the brick level might be two small planes. Their time extent would be chosen to encompass the slowest waves, and their spatial extent could be two or three points, representing the idea that normally we can listen to only one person at a time, occasionally we can listen to two, and we can never listen to three people talking at the same time.

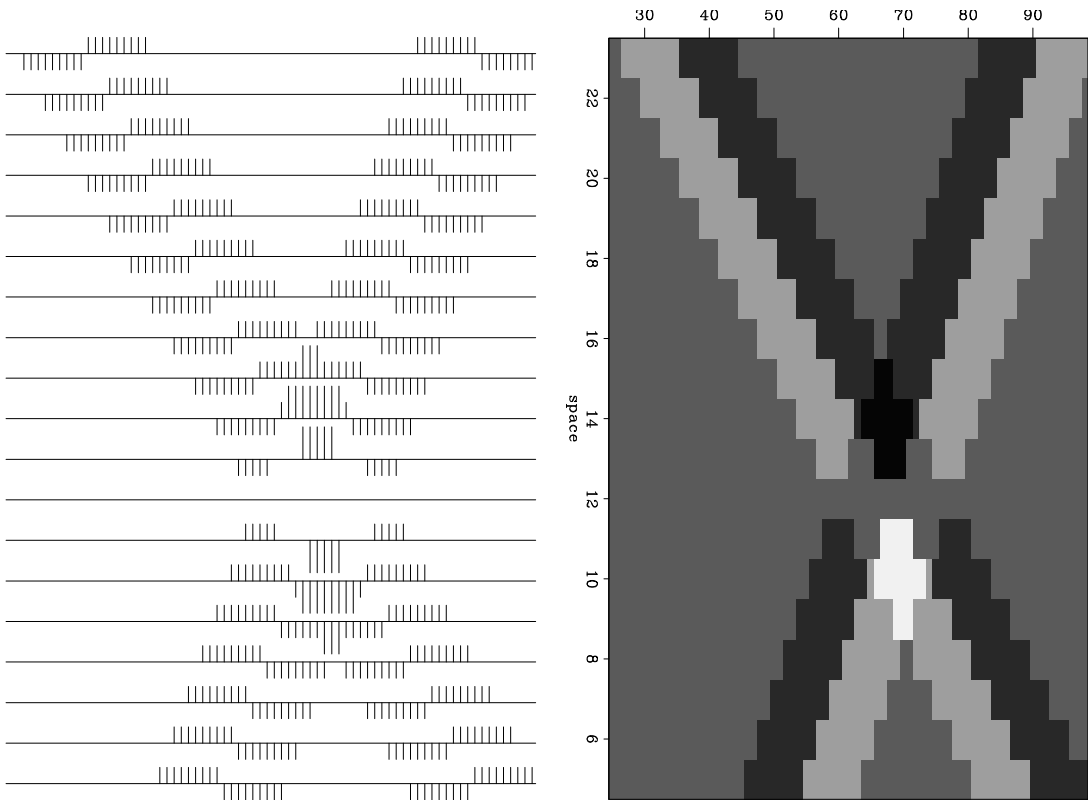
10.2. WAVE INTERFERENCE AND TRACE SCALING

Although neighboring seismometers tend to show equal powers, the energy on one seismometer can differ greatly from that of a neighbor for both theoretical reasons and practical ones. Should a trace ever be rescaled to give it the same energy as its neighbors? Here we review the strong theoretical arguments against rescaling. In practice, however, especially on land where coupling is irregular, scaling seems a necessity. The question is, what can go wrong if we scale traces to have equal energy, and more basically, where the proper scale factor cannot be recorded, what

should we do to get the best scale factor? A related question is how to make good measurements of amplitude versus offset. To understand these issues we review the fundamentals of wave interference.

Theoretically, a scale-factor problem arises because *locally*, wavefields, not energies, add. Nodes on standing waves are familiar from theory, but they could give you the wrong idea that the concept of node is one that applies only with sinusoids. Actually, destructive interference arises anytime a polarity-reversed waveform bounces back and crosses itself. Figure 10.3 shows two waves of opposite polarity crossing each other. Observe that one seismogram has a zero-valued signal, while its neighbors have anomalously higher amplitudes and higher energies than are found far away from the interference. The situation shown in Figure 10.3 does not occur easily in nature. Reflection naturally comes to mind, but usually the reflected wave crosses the incident wave at a later time and then they don't extinguish. Approximate extinguishing occurs rather easily when waves are quasi-monochromatic. We will soon see, however, that methodologies for finding scales all begin with deconvolution and that eliminates the monochromatic waves.

Figure 10.3: Superposition of plane waves of opposite polarity. lmn-super90 [ER]



10.2.1. Computing the proper scale factor for a seismogram

With data like Figure 10.3, rescaling traces to have equal energy would obviously be wrong. The question is, "How can we determine the proper scale factor?" As we have seen, a superposition of N plane waves exactly satisfies an N-th order (in x) difference equation. Given a 2-D wave field, we can find its PEF by minimizing output power. Then we ask the question, could rescaling the traces give a lower output power? To answer this, we set up an optimization goal: Given the leveler (be it a cubic PEF or two planar ones), find the best trace scales. (After solving this, we could return to re-estimate the leveler, and iterate.) To solve for the scales, we need a subroutine that scales traces and the only tricky part is that the adjoint should bring us back to the space of scale factors. This is done by `scaletrace` scaletrace Notice that to estimate scales, the adjoint forms an inner product of the raw data on the previously scaled data. Let the operator implemented by `scaletrace` be denoted by **D**, which is mnemonic for "data" and for "diagonal matrix," and let the vector of scale factors be denoted by **s** and the leveler by **A**. Now we consider the fitting goal $\mathbf{0} \approx \mathbf{ADs}$. The trouble with this fitting goal is that the solution is obviously $\mathbf{s} = \mathbf{0}$. To avoid the trivial solution $\mathbf{s} = \mathbf{0}$, we can choose from a variety of supplemental fitting goals. One possibility is that for the i-th scale factor we could add the fitting goal

```
module scaletrace {
integer, private                :: n1, n2
real, dimension( :, :), pointer :: data
#%  _init( data)
n1 = size( data, 1)
n2 = size( data, 2)
#% _lop( scale( n2), sdata( n1, n2))
integer i1,i2
do i2= 1, n2
do i1= 1, n1
        if( adj)
                scale(   i2) += sdata(i1,i2) * data(i1,i2)
        else
                sdata(i1,i2) += scale(   i2) * data(i1,i2)
}
```

Back

$s_i \approx 1$. Another possibility, perhaps better if some of the signals have the opposite of the correct polarity, is that the sum of the scales should be approximately unity. I regret that time has not yet allowed me to identify some interesting examples and work them through.

10.3. LOCAL MONOPLANE ANNIHILATOR

LOMOPLAN (LOcal MOno PLane ANnihilator) is a data-adaptive filter that extinguishes a local monoplane, but cannot extinguish a superposition of several planes. We presume an ideal sedimentary model as made of (possibly curved) parallel layers. Because of the superposition principle, data can be a superposition of several plane waves, but the ideal model should consist locally of only a single plane. Thus, LOMOPLAN extinguishes an ideal model, but not typical data. I conceived of LOMOPLAN as the "ultimate" optimization criterion for inversion problems in reflection seismology (1992b) but it has not yet demonstrated that it can attain that lofty goal. Instead, however, working in two dimensions, it is useful in data interpretation and in data quality inspection.

The main way we estimate parameters in reflection seismology is that we max-

imize the coherence of theoretically redundant measurements. Thus, to estimate velocity and statics shifts, we maximize something like the power in the stacked data. Here I propose another optimization criterion for estimating model parameters and missing data. An interpreter looking at a migrated section containing two dips in the same place suspects wave superposition more likely than bedding texture superposition. To minimize the presence of multiple dipping events in the same place, we should use the mono plane annihilator (MOPLAN) filter as the weighting operator for any fitting goal. Because the filter is intended for use on images or migrated data, not on data directly, I call it a *plane* annihilator, not a *planewave* annihilator. (A time-migration or merely a stack, however, might qualify as an image.) We should avoid using the word "wavefront" because waves readily satisfy the superposition principle, whereas images do not, and it is this aspect of images that I advocate and formulate as "prior information."

An example of a MOPLAN in two dimensions, $(\partial_x + p_x \partial_\tau)$, is explored in Chapter 4 of PVI (Claerbout, 1992a), where the main goal is to estimate the (τ, x)-variation of p_x. Another family of MOPLANs arise from multidimensional prediction-error filtering described earlier in this book and in PVI, Chapter 8.

Here I hypothesize that a MOPLAN may be a valuable weighting function for

many estimation problems in seismology. Perhaps we can estimate statics, interval velocity, and missing data if we use the principle of minimizing the power out of a LOcal MOno PLane ANnihilator (LOMOPLAN) on a migrated section. Thus, those embarrassing semicircles that we have seen for years on our migrated sections may hold one of the keys for unlocking the secrets of statics and lateral velocity variation. I do not claim that this concept is as powerful as our traditional methods. I merely claim that we have not yet exploited this concept in a systematic way and that it might prove useful where traditional methods break.

For an image model of nonoverlapping curved planes, a suitable choice of weighting function for fitting problems is the local filter that destroys the best fitting local plane.

10.3.1. Mono-plane deconvolution

The coefficients of a 2-D monoplane annihilator filter are defined to be the same as those of a 2-D PEF of spatial order unity; in other words, those defined by either (10.1) or (10.2). The filter can be lengthened in time but not in space. The choice

of exactly two columns is a choice to have an analytic form that can exactly destroy a single plane, but cannot destroy two. Applied to two signals that are statistically independent, the filter (10.2) reduces to the well-known prediction-error filter in the left column and zeros in the right column. If the filter coefficients were extended in both directions on t and to the right on x, the two-dimensional spectrum of the input would be flattened.

10.3.2. Monoplanes in local windows

The earth dip changes rapidly with location. In a small region there is a local dip and dip bandwidth that determines the best LOMOPLAN (LOcal MOPLAN). To see how to cope with the edge effects of filtering in a small region, and to see how to patch together these small regions, recall subroutine `patchn()` /prog:patch and the weighting subroutines that work with it.

Figure 10.4 shows a synthetic model that illustrates local variation in bedding. Notice dipping bedding, curved bedding, unconformity between them, and a fault in the curved bedding. Also, notice that the image has its amplitude tapered to zero on the left and right sides. After local monoplane annihilation (LOMOPLAN), the

continuous bedding is essentially gone. The fault and unconformity remain.

> The local spatial prediction-error filters contain the essence of a factored form of the inverse spectrum of the model.

Because the plane waves are local, the illustrations were made with module `lopef` /prog:lopef.

10.3.3. Crossing dips

Figure 10.5 deserves careful study. The input frame is dipping events with amplitudes slowly changing as they cross the frame. The dip of the events is not commensurate with the mesh, so we use linear interpolation that accounts for the irregularity along an event. The output panel tends to be small where there is only a single dip present. Where two dips cross, they tend to be equal in magnitude. Studying the output more carefully, we notice that of the two dips, the one that is strongest on the input becomes irregular and noisy on the output, whereas the other dip tends to remain phase-coherent.

I could rebuild Figure 10.5 to do a better job of suppressing monodip areas if I

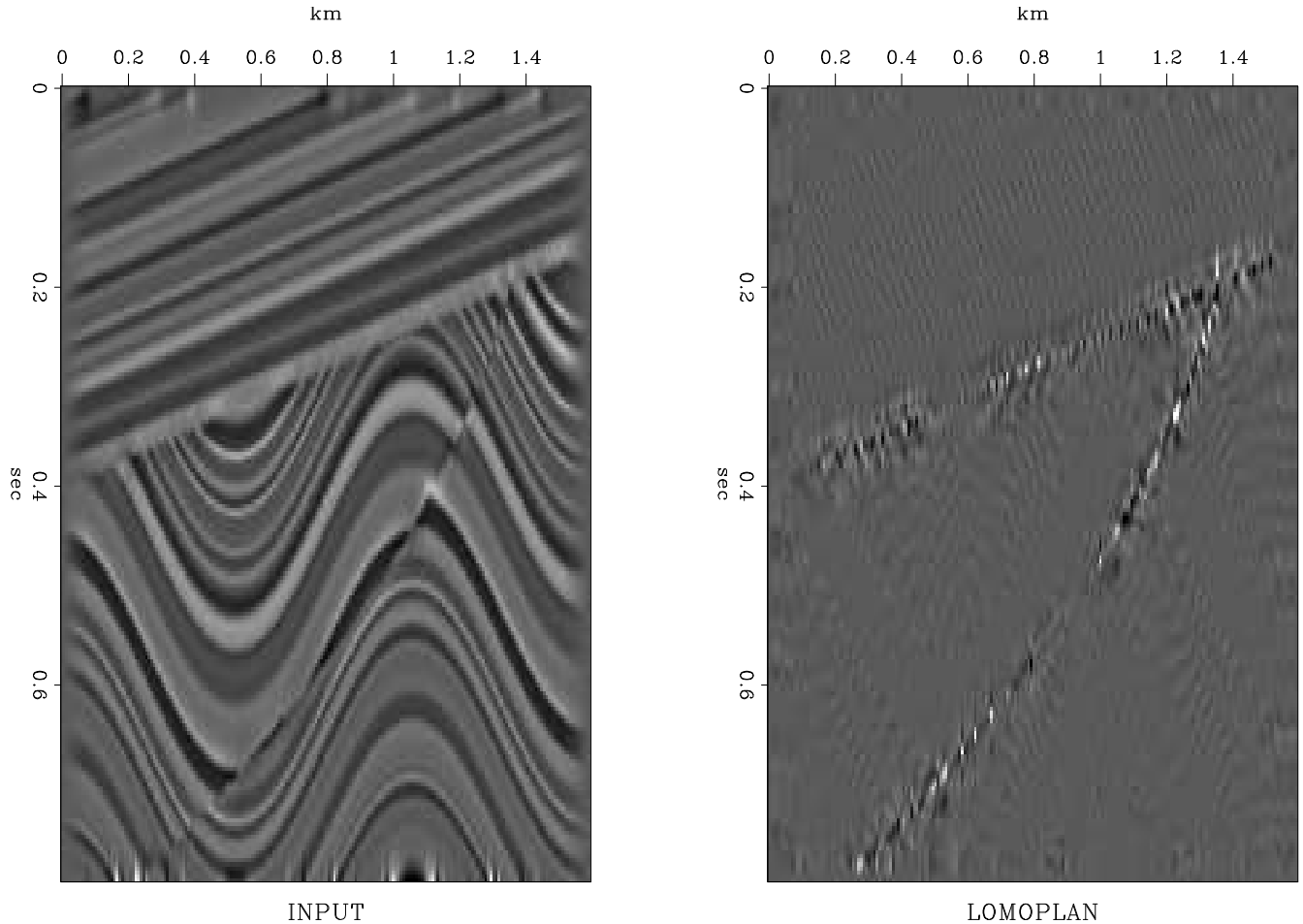


Figure 10.4: Left is a synthetic reflectivity model. Right is the result of local monoplane annihilation. lmn-sigmoid090 [ER]

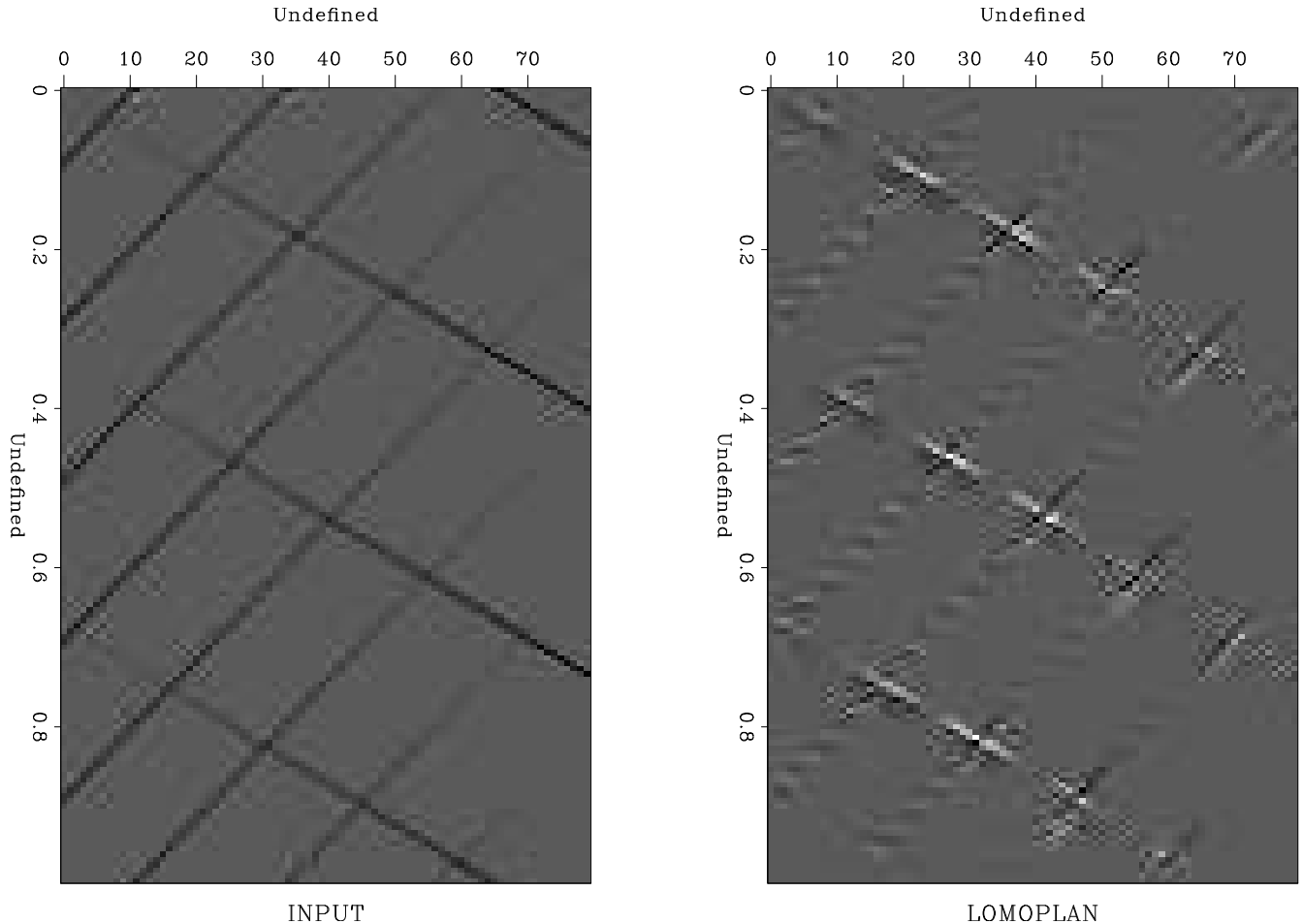


Figure 10.5: Conflicting dips before and after application of a local monoplane annihilator. Press button for movie. The movie sequence is: 1: data, 2: data after LOMOPLAN, 3: like previous but windows not overlapping, 4: predicted data lmn-conflict90 [ER]

passed the image through a lowpass filter, and then designed a gapped deconvolution operator. Instead, I preferred to show you high-frequency noise in the place of an attenuated wavefront.

The residual of prediction-error deconvolution tends to have a white spectrum in time. This aspect of deconvolution is somewhat irritating and in practice it requires us to postfilter for display, to regain continuity of signals. As is well known (PVI, for example), an alternative to postfiltering is to put a gap in the filter. A gapped filter should work with 2-D filters too, but it is too early to describe how experimenters will ultimately choose to arrange gaps, if any, in 2-D filters. There are some interesting possibilities. (Inserting a gap also reduces the required number of CD iterations.)

10.3.4. Tests of 2-D LOMOPLAN on field data

Although the LOMOPLAN concept was developed for geophysical *models*, not raw *data*, initial experience showed that the LOMOPLAN program is effective for quality testing data and data interpretation.

Some field-data examples are in Figures 10.6 and 10.7. These results are not

surprising. A dominant local plane is removed, and noise or the second-from-strongest local plane is left. These data sets fit the local plane model so well that subtracting the residual noise from the data made little improvement. These figures are clearer on a video screen. To facilitate examination of the residual on Figure 10.6 on paper (which has a lesser dynamic range than video), I recolored the white residual with a short triangle filter on the time axis. The residual in Figure 10.7 is large at the dead trace and wherever the data contains crossing events. Also, closer examination showed that the strong residual trace near 1.1 km offset is apparently slightly time-shifted, almost certainly a cable problem, perhaps resulting from a combination of the stepout and a few dead pickups. Overall, the local-plane residual shows a low-frequency water-velocity wave seeming to originate from the ship.

10.4. GRADIENT ALONG THE BEDDING PLANE

The LOMOPLAN (LOcal MOnoPLane ANnihilator) filter in three dimensions is a deconvolution filter that takes a volume in and produces two volumes out. The x-output volume results from a first order prediction-error filter on the x-axis, and the y-output volume is likewise on the y-axis.

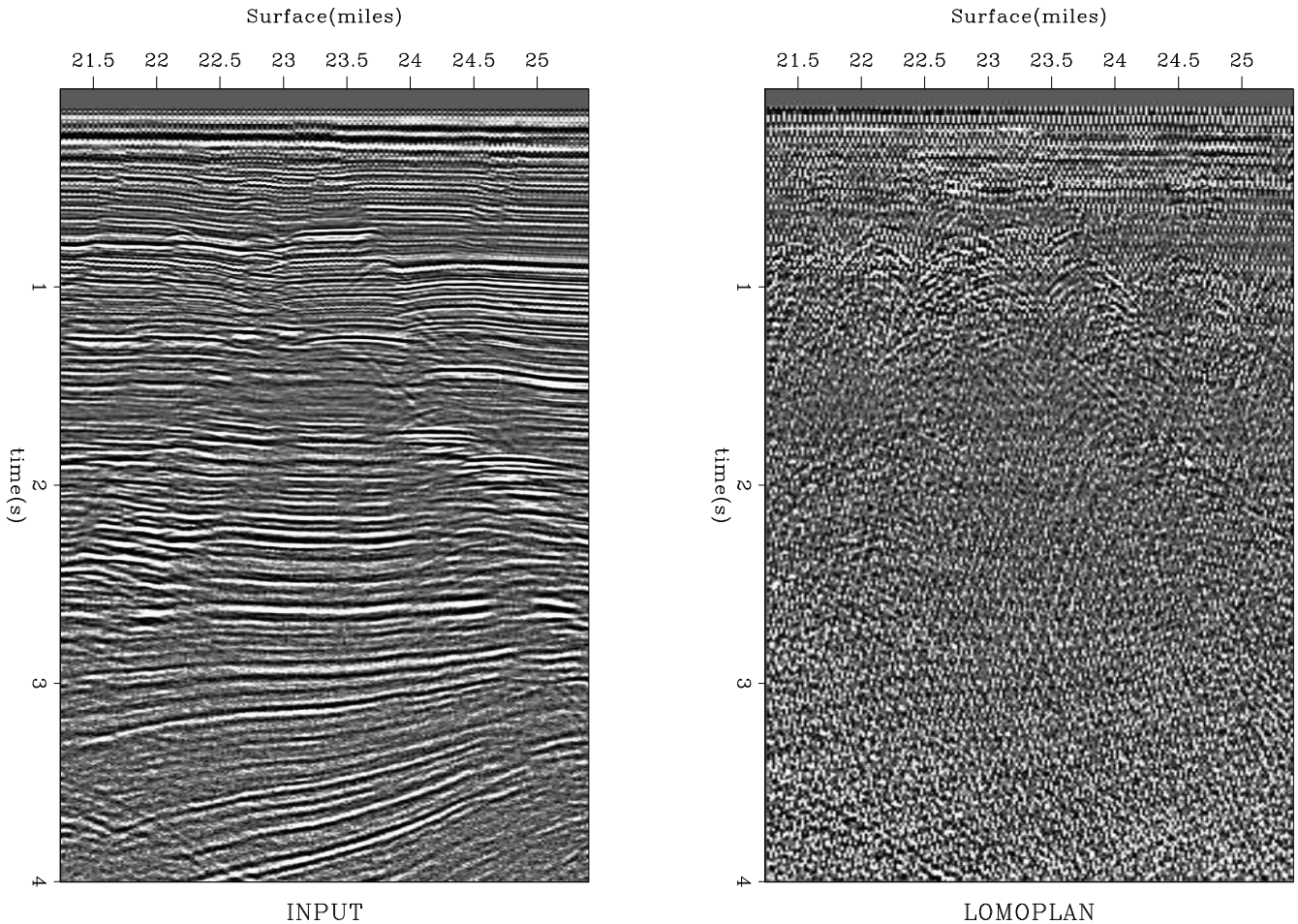


Figure 10.6: Data section from the Gulf of Mexico (left) and after LOMOPLAN (right) Press button for movie. lmn-dgulf90 [ER]

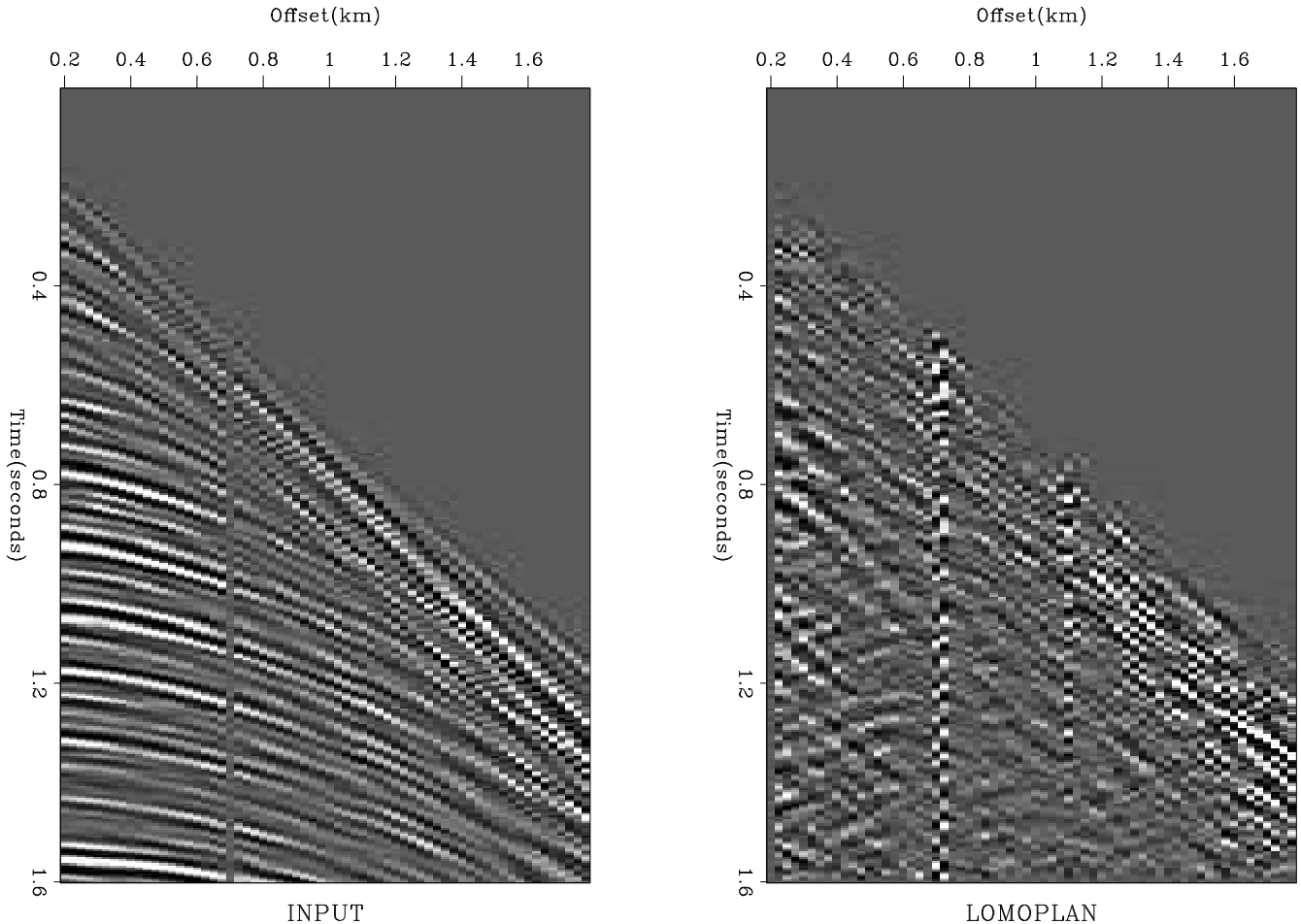


Figure 10.7: Portion of Yilmaz and Cumro data set 27 (left) and after LOMOPLAN (right). Press button for movie. lmn-yc2790 [ER]

Although I conceived of 2-D LOMOPLAN as the "ultimate" optimization criterion for inversion problems in reflection seismology of sedimentary sections, it turned out that it was more useful in data interpretation and in data-quality inspection. In this study, I sought to evaluate usefulness with *three*-dimensional data such as 3-D stacks or migrated volumes, or 2-D prestack data.

In experimenting with 3-D LOMOPLAN, I came upon a conceptual oversimplification, which although it is not precisely correct, gives a suitable feeling of the meaning of the operator. Imagine that the earth was flat horizontal layers, except for occasional faults. Then, to find the faults you might invoke the horizontal gradient of the 3-D continuum of data. The horizontal components of gradient vanish except at a fault, where their relative magnitudes tell you the orientation of the fault. Instead of using the gradient vector, you could use prediction-error filters of first order (two components) along x and y directions. 3-D LOMOPLAN is like this, but the flat horizontal bedding may be dipping or curved. No output is produced (ideally) except at faults. The 3-D LOMOPLAN is like the gradient *along the plane of the bedding*. It is nonzero where the bedding has an intrinsic change.

LOMOPLAN flags the bedding where there is an intrinsic change.

10.4.1. Definition of LOMOPLAN in 3-D

Three-dimensional LOMOPLAN is somewhat like multiple passes of two-dimensional LOMOPLAN; i.e., we first LOMOPLAN the (t,x)-plane for each y, and then we LOMOPLAN the (t,y)-plane for each x. Actually, 3-D LOMOPLAN is a little more complicated than this. Each LOMOPLAN filter is designed on all the data in a small (t,x,y) volume.

To put the LOcal in LOMOPLAN we use subcubes (bricks). Recall that we can do 2-D LOMOPLAN with the prediction-error subroutine `find_lopef()` /prog:lopef. To do 3-D LOMOPLAN we need to make two calls to subroutine `find_lopef()`, one for the x-axis in-line planar filters and one for the y-axis crossline filters. That is what I will try next time I install this book on a computer with a bigger memory.

10.4.2. The quarterdome 3-D synthetic (qdome)

Figure 10.4 used a model called "Sigmoid." Using the same modeling concepts, I set out to make a three-dimensional model. The model has horizontal layers near the top, a Gaussian appearance in the middle, and dipping layers on the bottom, with horizontal unconformities between the three regions. Figure 10.8 shows a vertical

slice through the 3-D "qdome" model and components of its LOMOPLAN. There is also a fault that will be described later. The most interesting part of the qdome model is the Gaussian center. I started from the equation of a Gaussian

$$z(x,y,t) \quad = \quad e^{-(x^2+y^2)/t^2} \tag{10.3}$$

and backsolved for t

$$t(x,y,z) \quad = \quad \sqrt{\frac{x^2+y^2}{-\ln z}} \tag{10.4}$$

Then I used a random-number generator to make a blocky one-dimensional impedance function of t. At each (x,y,z) location in the model I used the impedance at time $t(x,y,z)$, and finally defined reflectivity as the logarithmic derivative of the impedance. Without careful interpolation (particularly where the beds pinch out) a variety of curious artifacts appear. I hope to find time to use the experience of making the qdome model to make a tutorial lesson on interpolation. A refinement to the model is that within a certain subvolume the time $t(x,y,z)$ is given a small additive constant. This gives a fault along the edge of the subvolume. Ray Abma defined the subvolume for me in the qdome model. The fault looks quite realistic, and it is

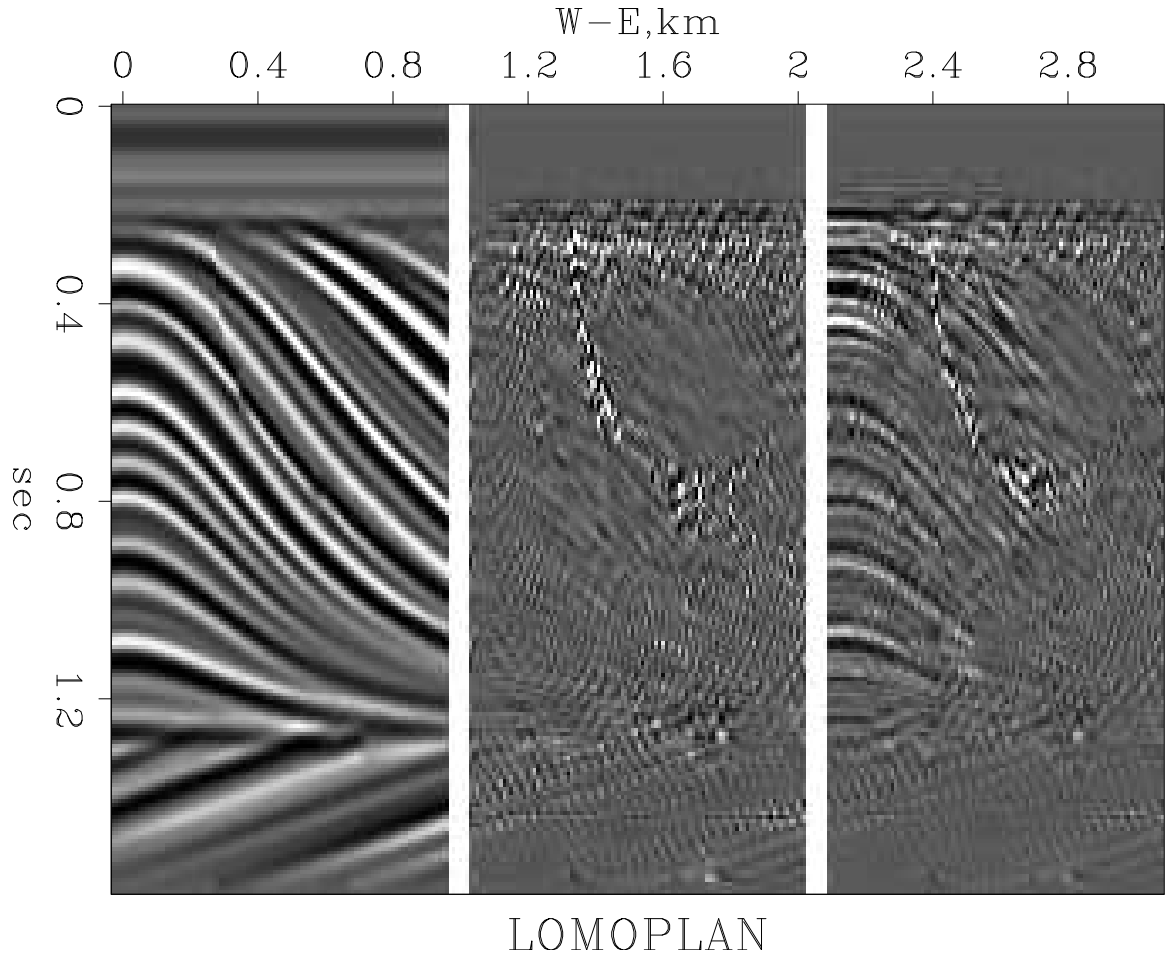


Figure 10.8: Left is a vertical slice through the 3-D “qdome” model. Center is the in-line component of the LOMOPLAN. Right is the cross-line component of the LOMOPLAN. lmn-qdomesico90 [CR]

easy to make faults of any shape, though I wonder how they would relate to realistic fault dynamics. Figure 10.9 shows a top view of the 3-D qdome model and components of its LOMOPLAN. Notice that the cross-line spacing has been chosen to be double the in-line spacing. Evidently a consequence of this, in both Figure 10.8 and Figure 10.9, is that the Gaussian dome is not so well suppressed on the crossline cut as on the in-line cut. By comparison, notice that the horizontal bedding above the dome is perfectly suppressed, whereas the dipping bedding below the dome is imperfectly suppressed.

Finally, I became irritated at the need to look at *two* output volumes. Because I rarely if ever interpreted the polarity of the LOMOPLAN components, I formed their sum of squares and show the single square root volume in Figure 10.10.

10.5. 3-D SPECTRAL FACTORIZATION

Hi Sergey, Matt, and Sean, Here are my latest speculations, plans: The 3-D Lomoplan resembles a gradient, one field in, two or three out. Lomoplan times its adjoint is like a generalized laplacian. Factorizing it yields a lomoplan generalization of the helix derivative, i.e. a one-to-one operator with the same spectral charactoristic as

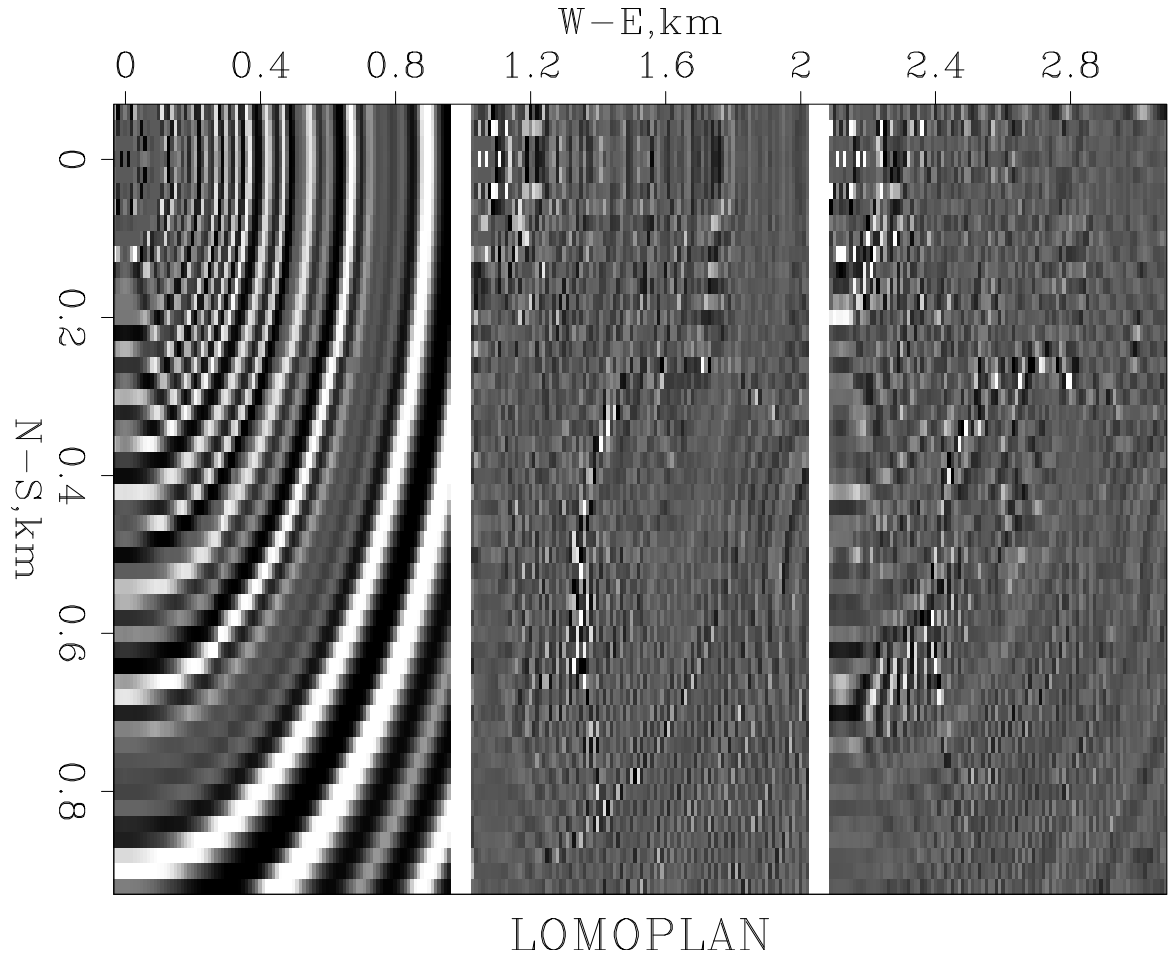


Figure 10.9: Left is a horizontal slice through the 3-D qdome model. Center is the in-line component of the LOMOPLAN. Right is the cross-line component of the LOMOPLAN. Press button for volume view. lmn-qdometoco90 [CR]

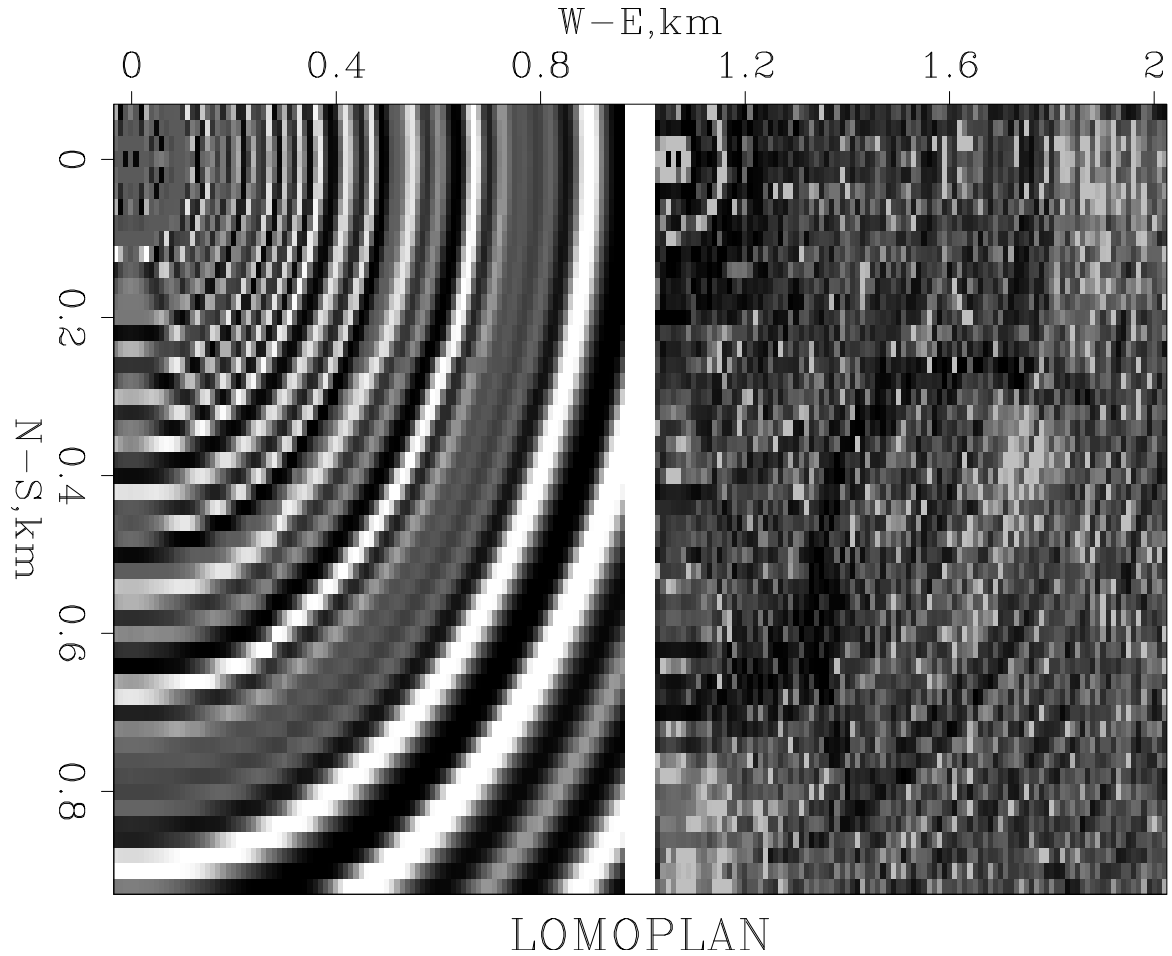


Figure 10.10: Left is the model. Right is the magnitude of the LOMOPLAN components in Figure 10.9. Press button for volume view. lmn-qdometora90 [CR]

the original lomoplan. It will probably not come out to be a juxtaposition of planes, will be more cube like. The advantage of being one-to-one is that it can be used as a preconditioner. The application, naturally enough, is estimating things with a prescribed dip spectrum. Things like missing data and velocities. Why use multiplanar lomoplan estimates if they will then be converted by this complicated process into a cube? Why not estimate the cube directly? Maybe to impose the “pancake" model instead of the noodle model of covariance. Maybe to reduce the number of coefficients to estimate. I haven’t figured out yet how to convert this speculation into an example leading to some figures. If you like the idea, feel free to beat me to it :)

REFERENCES

Claerbout, J. F., 1992a, Earth Soundings Analysis: Processing Versus Inversion: Blackwell Scientific Publications. 677

Claerbout, J. F., 1992b, Information from smiles: Mono-plane-annihilator weighted regression: SEP–**73**, 409–420. 676

Chapter 11

Some research examples

SEP students and researchers have extended the work described in this book. A few of their results are summarized here without the details and working codes.

11.1. GULF OF MEXICO CUBE

David Lumley from Chevron gave James Rickett some nice 3-D data from the Gulf of Mexico. There movie shows time slices at intervals of about 6ms. These slices are about 18 feet apart. That is about 7,000 years of deposition in the Gulf of Mexico. Altogether it is about a million years (about the age of the human species). Figure 11.1 shows some nice time slices.

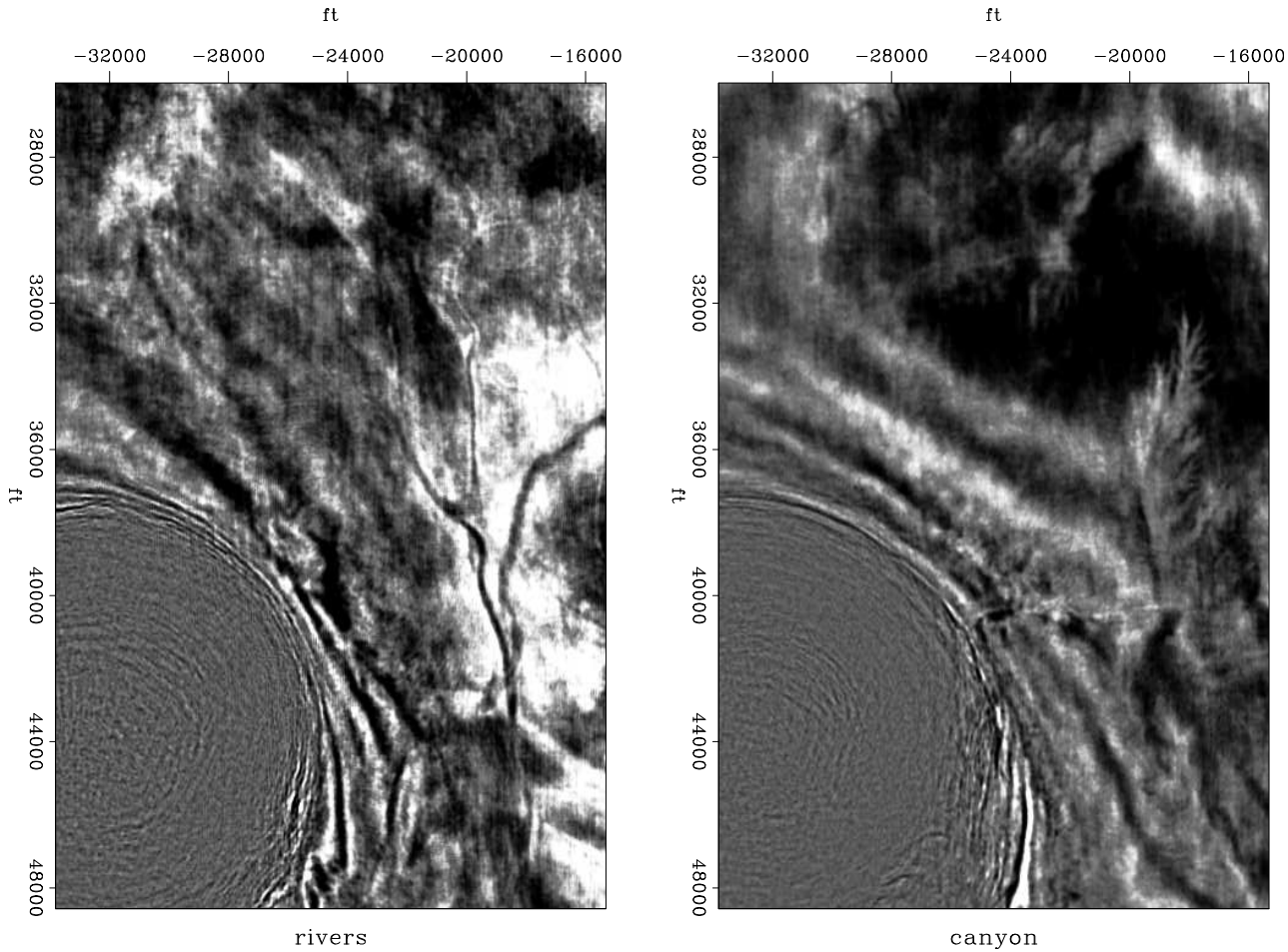


Figure 11.1: Some time slices show a salt dome, some river channels, a dendritic drainage canyon, and a fault. Press button for interactive "Rickmovie". rez-canyon [ER]

Chapter 12

SOFTWARE SUPPORT

“Ratfor" (RATional FORtran) is a dialect of Fortran that is more concise than raw Fortran. Our present Ratfor “compiler,” `ratfor90`, is a simple word-processing pro-

gram (written[1] in Perl and freely distributed) that inputs an attractive Fortran-like dialect and outputs Fortran90. Mainly, the word-processor produces Fortran statements like `end do`, `end if`, `end program` and `end module`, from the Ratfor "}". Ratfor source is about 25-30% smaller than the equivalent Fortran, so it is equivalently more readable.

Bare-bones Fortran is our most universal computer language for computational physics. For general programming, however, it has been surpassed by C. Ratfor is Fortran with C-like syntax. Ratfor was invented by the people[2] who invented C. After inventing C, they realized that they had made a mistake (too many semicolons) and they fixed it in Ratfor, although it was too late for C. Otherwise, Ratfor uses C-like syntax, the syntax that is also found in the popular languages C++ and Java.

At SEP we supplemented Ratfor77 by preprocessors to give Fortran77 the ability to allocate memory on the fly. These abilities are built into Fortran90 and are seamlessly included in Ratfor90. To take advantage of Fortran90's new features while maintaining the concise coding style provided by Ratfor, we had to write a

---

[1] The Ratfor90 preprocessor was written by my colleague, Bob Clapp.

[2] Kernighan, B.W. and Plauger, P.J., 1976, Software Tools: Addison-Wesley.

new Ratfor preprocessor, Ratfor90, which produces Fortran90 rather than Fortran77 code.

You should be able to read Ratfor if you already know Fortran or any similar computer language. Writing Ratfor is easy if you already know Fortran because written Fortran is valid Ratfor. You can mix Ratfor and Fortran. The Ratfor processor is not a compiler but a simple word-processing program which passes Fortran (which it does not understand) through unchanged. The Ratfor processor converts the Ratfor dialect to Fortran. To maximize the amount of Ratfor, you will need to know its rules. Here they are:

Statements on a line may be separated by ";". Statements may be grouped together with braces { }. Do loops do not require statement numbers because { } defines the range. Given that `if( )` is true, the statements in the following { } are done. `else{ }` does what you expect. We may *not* contract `else if` to `elseif`. We may omit the braces { } where they contain only one statement. `break` (equivalent to the Fortran90 `exit`) causes premature termination of the enclosing { }. `while( ) { }` repeats the statements in { } while the condition ( ) is true. Ratfor recognizes `repeat { ... } until( )` as a loop that tests at the bottom. `next` causes skipping to the end of any loop and a retrial of the test condition. `next` (equivalent

to the Fortran90 `cycle` statement) is rarely used, and the Ratfor90 coder may write either `next` or `cycle`. Here we encounter an inconsistency between Fortran and C-language. Where Ratfor uses `next`, the C-language uses `continue` (which in Ratfor and Fortran is merely a place holder for labels). The Fortran relational operators `.gt.`, `.ge.`, `.ne.`, etc. may be written >, >=, !=, etc. The logical operators `.and.` and `.or.` may be written && and ||. Anything from a # to the end of the line is a comment. A line may be continued in Ratfor by ending it with the underscore charactor "_" (like Fortran90's &).

Indentation in Ratfor is used for readability. It is not part of the Ratfor language. Choose your own style. I have overcondensed. There are two pitfalls associated with indentation. The beginner's pitfall is to assume that a `do` loop ends where the indentation ends. The loop actually ends after the first statement. A larger scope for the `do` loop is made by enclosing multiple statements in braces. The other pitfall arises in any construction like `if() ... if() ... else`. The `else` goes with the last `if()` regardless of indentation. If you want the `else` with the earlier `if()`, you must use braces like `if() { if() ... } else ...`. Ratfor also recognizes the looping statement used in C, C++, and Java. It is `for(`initialize`;` condition`;` reinitialize`) { }`.

12.0.1. Changes and backward compatibility

We were forced to make one change to Ratfor90 because of new things in Fortran90. Ratfor77 allows & and | for the logical operators && and ||. While attractive, it is not part of the C family of languages and we had to drop it because Fortran90 adopts & for line continuation.

Because we are not compiler writers, we dropped a rarely used feature of Ratfor77 that was not easy for us to implement and is ugly anyway: Ratfor77 recognizes `break 2` which escapes from {{ }}.

Changing all the code that generated illustrations for four textbooks (of various ages) also turned up a few more issues: Fortran90 uses the words `scale` and `matmul` as intrinsics. Old Fortran77 programs using those words as variable names must be changed. Ratfor77 unwisely allowed variables of intrinsic (undeclared) types. We no longer allow this. Ratfor90 forces `implicit none`.

New features in Ratfor90 are bracketed type, subroutine, function, and module procedures. In some ways this a further step towards the C, C++, Java model. It makes complicated modules, subroutines inside subroutines, and other advanced features of Fortran90 easier to interpret. Ratfor90 has better error messages than Ratfor77. Besides the use of stderr, a new file (`ratfor_problem`) marks the diffi-

culty.

12.0.2. Examples

Below are simple Ratfor subroutines for erasing an array (`zero()`); (`null()`); for copying one array to another (`copy()`); for the signum function $sgn(x) = x/|x|$ (`signum()`); and (`tcaf`), a program using fortran90 modules and overloading to transient convolution.

12.0.3. Memory allocation in subroutines

For backward compatibility we allow the "temporary" memory allocation introduced by our Ratfor77 processor for example:

```
temporary real*4 data(n1,n2,n3), convolution(j+k-1)
```

These declarations must follow other declarations and precede the executable statements. Automatic arrays are supported in Fortran90. To allow full code compatibility, `Ratfor90` simply translates this statement to

```
real*4 data(n1,n2,n3), convolution(j+k-1).
```

12.0.4. The main program environment

`Ratfor90` includes some traditional SEP local-memory-allocation and data-base-I/O statements that are described below. It calls an essential seplib initialization routine `initpar()`, organizes the self-doc, and simplifies data-cube input. The basic syntax for memory allocation is `allocate: real x(n1,n2)`. `Ratfor90` translates this syntax into a call to dynamically allocate a `allocatable` array. See the on-line self-documentation or the manual pages for full details. Following is a complete Ratfor program for a simple task:

```
# <in.H  Scale scaleval=1. > out.H
#
#       Copy input to output and scale by scaleval
# keyword generic scale
#%
integer n1, n2, n3, esize
from history:   integer n1, n2, n3, esize
if (esize !=4)  call erexit('esize != 4')
allocate:       real x(n1,n2)
subroutine scaleit( n1,n2, x)
```

```
integer i1,i2, n1,n2
real    x(n1,n2), scaleval
from par:       real scaleval=1.
call hclose()                   # no more parameter handling.
call sreed('in', x, 4*n1*n2)
do i1=1,n1
        do i2=1,n2
                x(i1,i2) = x(i1,i2) * scaleval
call srite( 'out', x, 4*n1*n2)
return;         end
```

12.1. SERGEY'S MAIN PROGRAM DOCS

Many of the illustrations in this book are made with main programs that can be reused (in the SEP environment) for other applications. Here is a summary of their documentation.

12.1.1. Autocorr - compute autocorrelation for helix filters

```
Autocorr < filt.H > autocorr.H
```

Reads a helix filter. Outputs the positive lag of its autocorrelation (no space wasted).

from/to history	**integer**	*n1*	filter size
	integer array	*lag*	comma-separated list o
	real	*a0*=1	zero-lag coefficient

Modules: *helix.r90, autocorr.r90*

12.1.2. Bin2 - nearest neighbor binning in 2-D

```
Bin2 < triplets.H > map.H
```

Bins (x,y,z) data triplets. Normalizes by bin fold.

from history	**integer**	*n1*, *n2*	*n1* is number of triplets
from par	**integer**	*n1*, *n2* – map size	
	real	*o1*, *o2*, *d1*, *d2* – map dimensions	

Modules: *bin2.lop*

12.1.3. Conv - convolve two helix filters

```
Conv < filt1.H other=filt2.H > conv.H
```

Outputs the convolution of filt1 and filt2.

from/to history	**integer**	*n1*	filter size
	integer array	*lag*	comma-separated list o

Modules: *helix.r90, conv.r90*

12.1.4. Decon - Deconvolution (N-dimensional)

```
Decon < data.H filt= predictive=0 > decon.H
```

Deconvolution: predictive, Lomoplan, steep dip. Uses the helix and patching technology.

from history	**integer array**	*n*	*n1*, *n2*, *n3*, etc
from par	**filename**	*filt*	helix-type local PEF
	logical	*predictive*=0	predictive deconvolutio
	integer	*rect1* (optional)	smoothing on the first a
from aux (filt)	**integer**	*dim*	number of dimensions
	integer array	*w*	patch size
	integer array	*k*	number of windows

Modules: *tent.r90*, *patching.r90*, *loconvol.r90* , *helix.r90 triangle.r90*
See also: `Lopef`, `Helicon`

12.1.5. Devector - create one output filter from two input filters

```
Devector < filt1.H other=filt2.H > filt.H
```

Uses Wilson's factorization. filt = sqrt (filt1**2 + filt2**2)

from history and from aux (other)	**integer**	*n1*	number of filter coeffic
	integer arra	*n2*,*n3*,...	number of filters
	integer array	*lag*	helix filter lags
from par	**integer**	*niter*=20	number of WIlson's ite
	integer	*n1*	number of output filter
	integer array	*lag*	output lags

Modules: *wilson.r90*, *autocorr.r90*, *compress.r90*

12.1.6. Helderiv - Helix derivative filter in 2-D

```
Helderiv < in.H helix=1 na=16 > out.H
```

Factors the laplacian operator. Applies helix derivative. Loops over n3

from history	**integer**	*n1*, *n2*	
from par	**logical**	*helix*=1	if 0, apply the gradient
	integer	*na*=16	filter size (half the num
			cients)
	real	*eps*=0.001	zero frequency shift on

Modules: *helicon.lop*, *helderiv.r90*

12.1.7. Helicon - Helix convolution and deconvolution (N-dimensional!)

```
Helicon < in.H filt= adj=0 div=0 > out.H
```

Applies helix convolution (polynomial multiplication) or deconvolution (polynomial division). One is the exact inverse of the other. Watch for helical boundary conditions.

from history	**integer array**	*n*	reads *n1*, *n2*, *n3*, ...
from par	**filename**	*filt*	helix filter file
	integer	*adj*=0	apply adjoint (backwar
	integer	*div*=0	apply inverse recursive division)
from aux (filt)	**integer array**	*h*	helix grid (can be *h1*, *h*
	integer array	*lag*=1,...,n1	comma separated list o
	real	*a0*=1	zero-lag filter coefficie

Modules: *helicon.lop*, *polydiv.lop*, *regrid.r90*, *helix.r90*

12.1.8. Helocut - Helix Lowcut filter in 2-D

```
Helocut < in.H helix=1 na=16 eps=0.1 > out.H
```

Applies helix convolution with a low-cut factor, based on factoring the laplacian filter. Also loops over n3.

from history	**integer**	*n1*, *n2*	
from par	**logical**	*helix*=1	if 0, apply the gradient
	real	*eps*	sets the lowcut frequen
	integer	*na*=16	filter size (half the num cients)

Modules: *helicon.lop, helocut.r90*

12.1.9. Hole - Punch ellipsoidal hole in 2-D data

```
Hole < data.H > hole.H
```

Hole's dimensions and orientation are currently fixed

from history	**integer**	*n1*, *n2*

See also: `Make`

12.1.10. Igrad - take gradient on the first axis

```
Igrad < map.H > grad.H
```

Works on 2-D data, gradient is (1,-1) filter

from history	**integer**	*n1*, *n2*

Modules: *igrad1.lop*

12.1.11. LPad - Pad and interleave traces

```
LPad < small.H jump=2 mask=,> large.H
```

Each initial trace is followed by *jump* zero traces, the same for planes.

from history	**integer**	*n1*, *n2*, *n3*	
from par	**integer**	*jump*=2	how much to expand th
	filename	*mask*	selector for known trac
			put)
to history	**integer**	*n2*=n2*jump (if *n2* > 1), *n3*=n3*jump (if *n3* > 1)	

See also: `LPef`

12.1.12. LPef - Find PEF on aliased traces

```
LPef < in.H jump=2 a= center=1 gap=0 > out.H
```

Finds a prediction-error filter, assuming missing traces

from history	**integer array**	*n*	reads *n1*, *n2*, *n3*, etc.
from par	**integer**	*jump*=2	how much to expand th
	integer array	*a*=	PEF size
	integer array	*center*=1	PEF centering
	integer array	*gap*=0	PEF gapping

Modules: *lace.r90*, *helix.r90*, *print.r90*, *compress.r90*
See also: `Pef`

12.1.13. Lapfill2 - fill missing data by minimizing the Laplacian

```
Lapfill2 < map.H > filled.H
```

Works on 2-D data only.

from history	**integer**	*n1*, *n2*	
from par	**integer**	*niter*=200	number of CG iteration

Modules: *lapfill.r90*
See also: `Miss`, `MSMiss`

12.1.14. LoLPef - Find PEF on aliased traces (with patching)

```
LoLPef < in.H jump=2 a= center=1 gap=0 > out.H
```

Finds a prediction-error filter, assuming missing traces

from history	**integer array**	*n*	reads *n1*, *n2*, *n3*, etc.
from par	**integer array**	*w*=20,20,6	patch size
	integer array	*k* (optional)	number of windows
	integer	*jump*=2	how much to expand th
	integer array	*a*=	PEF size
	integer array	*center*=1	PEF centering
	integer array	*gap*=0	PEF gapping

Modules: *lolace.r90*

See also: `Pef`

12.1.15. Lomiss - Missing data interpolation with a prescribed helix filter

12.1.16. (in local patches)

```
Lomiss < in.H prec=1 niter=100 filt= [mask=] > interp.H
```

Fills missing data by mimimizing the data power after convolution. Works in any number of dimensions!

from history	**integer**	*n1*, *n2*, *n3*	
from par	**integer**	*prec*=1	use preconditioning for lation
	integer	*niter*=100	number of iterations
	filename	*filt*	helix filter
	filename	*mask* (optional)	selector for known data
from aux (sfilt, nfilt)	**integer**	*dim*	number of dimensions
	integer array	*w*	patch size
	integer array	*k*	number of windows

Modules: *lomis2.r90*, *helix.r90*, *tent.r90*

12.1.17. Lopef - Local Prediction-Error Filter (1-D, 2-D, and 3-D)

```
Lopef < data.H dim=3 steepdip=0 > pef.H
```

Local prediction-error filters are estimated with the helix and patching technology. Can also find filters for steep-dip deconvolution. Currently works in 1, 2, and 3 dimensions.

from history	**integer**	*n1, n2, n3*	
	real	*d1, d2, d3* (for steep-dip decon)	
from par	**integer**	*dim*=3	number of dimensions
	integer array	*w*=20,20,6	patch size
	integer array	*a*=5,2,1	filter size
	integer array	*k* (optional)	number of windows
	integer array	*gap*=0,0,0	filter gap
	integer array	*ctr* (optional)	filter centering
	logical	*steepdip*=0	steep-dip decon PEF
	real	*vel*=1.7	velocity for steep-dip d
	real	*tgap*=0.03	time gap for steep-dip
	filename	*mask* (optional)	data selector

Modules: *bound.r90, steepdip.r90, shape.r90, lopef.r90, print.r90, helix.r90*

See also: `Pef`, `Decon`

12.1.18. Losignoi - Local signal and noise separation (N-dimensional)

`Losignoi < data.H sfilt= nfilt= eps= > sign.H`

Signal and noise separation by inversion (super-deconvolution). Uses the helix and patching technologies.

from history	**integer array**	*n*	*n1*, *n2*, *n3*< etc
from par	**filename**	*sfilt*, *nfilt*	helix-type signal and n
	real	*eps*	the magic scaling para
	integer	*niter*=20	number of iterations
from aux (sfilt, nfilt)	**integer**	*dim*	number of dimensions
	integer array	*w*	patch size
	integer array	*k*	number of windows

Modules: *tent.r90*, *patching.r90*, *signoi.r90* , *helix.r90*
See also: `Decon`, `Lopef`, `Helicon`

12.1.19. MSHelicon - Multi-scale Helix convolution (N-dimensi

```
Helicon < in.H filt= ns= jump= adj=0 > out.H
```

Applies multiscale helix convolution.

from history	**integer array**	*n*	reads *n1*, *n2*, *n3*, ...
from par	**filename**	*filt*	helix filter file
	integer	*adj*=0	apply adjoint (backwar
	integer	*ns*	number of scales
	integer array	*jump*=0	filter scales
from aux (filt)	**integer array**	*h*	helix grid (can be *h1*, *h*
	integer array	*lag*=1,...,n1	comma separated list o
	real	*a0*=1	zero-lag filter coefficie

Modules: *mshelicon.lop*, *regrid.r90*, *mshelix.r90*

12.1.20. MSMiss - Multiscale missing data interpolation (N-dimensional)

```
MSMiss < in.H prec=1 niter=100 filt= [mask=] > interp.H
```

Fills missing data by mimimizing the data power after convolution.

from history	**integer array**	*n*	reads *n1*, *n2*, *n3*, ...
from aux (filt)	**integer**	*ns*	number of scales
	integer array	*jump*	comma separated list o
from par	**integer**	*prec*=1	use preconditioning fo
			lation
	integer	*niter*=100	number of iterations
	filename	*filt*	helix filter
	filename	*mask* (optional)	selector for known data

Modules: *msmis2.r90*, *mshelix.r90*, *bound.r90*

12.1.21. MSPef - Multi-scale PEF estimation

```
MSPef < in.H a= center= gap=0 ns= jump= [maskin=] [maskout=] > pef.H
```

Estimates a multi-scale PEF. Works in N dimensions

from history	**integer array**	*n*	reads *n1*, *n2*, *n3*
from par	**integer array**	*a*=	PEF size
	integer	*niter*=2*prod(a) (optional)	number of PEF iteratio
	integer array	*center*	PEF centering
	integer array	*gap*=0	PEF gapping
	integer	*ns*	number of scales
	integer array	*jump*	comma separated list o
	filename	*maskin*, *maskout* (optional)	data selectors

Modules: *mspef.r90*, *misinput.r90*, *mshelix.r90 createmshelixmod.r90*, *print.r90*
See also: `MSMiss Pef`

12.1.22. Make - generate simple 2-D synthetics with crossing plane waves

```
Make n1=100 n2=14 n3=1 n=3 p=3 t1=4 t2=4 > synth.H
```

Plane waves have fixed slopes, but random amplitudes

from par	**integer**	*n1*=100, *n2*=14, *n3*=1	data size
	integer	*n*=3	slope
	integer	*p*=3	power for generating ra
	integer	*t1*=3, *t2*=3	width of trinalge smoo

Modules: *triangle.lop*, *random.f90* (for compatibility with Fortran-77)

See also: `Hole`

12.1.23. Minphase - create minimum-phase filters

```
Minphase < filt.H niter=20 > minphase.H
```

Uses Wilson's factorization. The phase information is lost.

from history	**integer**	*n1*	number of filter coeffic
	integer array	*n2*,*n3*,...	number of filters
	integer array	*lag*	helix filter lags
from par	**integer**	*niter*=20	number of WIlson's ite

Modules: *wilson.r90*, *autocorr.r90*

12.1.24. Miss - Missing data interpolation with a prescribed helix filter

```
Miss < in.H prec=1 niter=100 padin=0 padout=0 filt= [mask=] > interp.H
```

Fills missing data by minimizing the data power after convolution. Works in any number of dimensions!

from history	**integer**	*n1*, *n2*, *n3*	
from par	**integer**	*prec*=1	use preconditioning for lation
	integer	*niter*=100	number of iterations
	integer	*padin*=0	pad data beginning
	integer	*padout*=0	pad data end
	filename	*filt*	helix filter
	filename	*mask* (optional)	selector for known data

Modules: *mis2.r90*, *bound.r90*, *helix.r90*

12.1.25. NHelicon - Non-stationary helix convolution and deconvolution

```
Helicon < in.H filt= adj=0 div=0 > out.H
```

Applies helix convolution (polynomial multiplication) or deconvolution (polynomial division). One is the exact inverse of the other. Watch for helical boundary conditions.

from history	**integer array**	*n*	reads *n1*, *n2*, *n3*, ...
from par	**filename**	*filt*	helix filter file
	integer	*adj*=0	apply adjoint (backwar
	integer	*div*=0	apply inverse recursive division)
from aux (filt)	**integer array**	*h*	helix grid (can be *h1*, *h*
	integer array	*lag*=1,...,n1	comma separated list o
	real	*a0*=1	zero-lag filter coefficie

Modules: *nhelicon.lop*, *npolydiv.lop*, *nhelix.r90*, *helix.r90*, *regrid.r90*

12.1.26. NPef - Estimate Non-stationary PEF in N dimensions

```
Pef < data.H a= center=1 gap=0 [maskin=] [maskout=] > pef.H
```

Estimates PEF by least squares, using helix convolution. Can ignore missing data

from history	**integer array**	*n*	reads *n1*, *n2*, *n3*, etc.
from par	**integer**	*niter*=100	number of iterations
	real	*epsilon*=0.01	regularization paramete
	integer array	*a*=	filter size
	integer array	*center*=1	zero-lag position (filter
	integer array	*gap*=0	filter gap
	filename	*maskin*, *maskout* (optional)	data selectors
to history	**integer array**	*lag*	comma separated list o

Modules: *nhelix.r90*, *createnhelixmod.r90*, *nmisinput.r90*, *npef.r90*,
See also: `MSPef`, `Pef`, `NHelicon`

12.1.27. Nozero - Read (x,y,z) data triples, throw out values of z > thresh, transpose

```
Nozero < triplets.H thresh=-210 > transp.H
```

The program is tuned for the Sea of Galilee data set

from history	**integer**	*n1*, *n2*	*n2* is the number of trip
	real	*thresh*=**-**210 - threshold (default is tuned for Galilee)	
to history	**integer**	*n1*, *n2*	*n1* is the number of thresh
		n2=3	

See also: `Bin2`

12.1.28. Parcel - Patching illustration

```
Parcel < in.H w= k= > out.H
```

Transforms data to patches and back without the weighting compenssation.

integer array	*w*	window size
integer array	*k*	number of windows in

Modules: *parcel.lop, cartesian.r90*

12.1.29. Pef - Estimate PEF in N dimensions

```
Pef < data.H a=[center=] [gap=] [maskin=] [maskout=] > pef.H
```

Estimates PEF by least squares, using helix convolution. Can ignore missing data

from history	**integer array**	*n*	reads *n1*, *n2*, *n3*, etc.
from par	**integer array**	*a*=	filter size
	integer	*niter*=2*prod(a) (optional)	number of
	PEF iterations		
	integer array	*center*=a/2+1 (optional)	zero-lag position (filter
	integer array	*gap*=0 (optional)	filter gap
	filename	*maskin*, *maskout* (optional)	data selectors
to history	**integer array**	*lag*	comma separated list o

Modules: *shape.r90*, *bound.r90*, *misinput.r90*, *pef.r90*, *compress.r90*, *print.r90*, *helix.r90*
See also: `MSPef`, `Fillmiss`, `Helicon`, `Decon`

12.1.30. Sigmoid - generate sigmoid reflectivity model

```
Sigmoid n1=400 n2=100 o1=0 d1=.004 o2=0 d2=.032 > synth.H
```

Sigmoid reflectivity model in 2-D: complex geological structure.

from par	**integer**	*n1*=400, *n2*=100	data size
	integer	*large*=5*n1	layering size
	real	*o1*=0, *d1*=0.004, *o2*=0., *d2*=0.032	grid spacing

Modules: *random.f90* (for compatibility with Fortran-77)
See also: `Make`

12.1.31. Signoi - Local signal and noise separation (N-dimensio

```
Signoi < data.H sfilt= nfilt= epsilon= > sig+noi.H
```

Signal and noise separation by optimization.

from history	**integer array**	*n*	*n1*, *n2*, *n3*< etc
from par	**filename**	*sfilt*, *nfilt*	helix-type signal and n
	real	*eps*	the magic scaling paran
	integer	*niter*=20	number of iterations

Modules: *signoi.r90* , *regrid.r90*
See also: `Losignoi`, `Pef`

12.1.32. Tentwt - Tent weight for patching

```
Tentwt dim=2 n= w= windwt= >wallwt.H
```

Computes the tent weight for patching.

from par	**integer**	*dim*=2	number of dimensions
	integer array	*n*	data size (n1, n2, etc)
	integer array	*w*	window size
	integer array	*k* (optional)	number of windows in
	integer array	*a* (optional)	window offset
	integer array	*center* (optional)	window centering

Modules: *tent.r90*, *wallwt.r90*

12.1.33. Vrms2int - convert RMS velocity to interval velocity

```
Vrms2int < vrms.H weight= vrms= niter= eps= > vint.H
```

Least-square inversion, preconditioned by integration.

from history	**integer**	*n1, n2*	
from par	**integer**	*niter*	number of iterations
	real	*eps*	scaling for precondition
	filename	*weight*	data weight for inversi
	filename	*vrms*	predicted RMS velocity

Modules: *vrms2int.r90*

12.1.34. Wilson - Wilson's factorization for helix filters

```
wilson < filt.H niter=20 [n1= lag=] > minphase.H
```

Reads a helix autocorrelation (positive side of it). Outputs its minimum-phase factor.

from/to history	**integer**	*n1*	filter size
	integer array	*lag*	comma-separated list o
	real	*a0*=1	zero-lag coefficient
from par	**integer**	*niter*=20	number of Newton's ite
	integer	*n1* (optional)	number of coefficients
	integer array	*lag* (optional)	comma-separated list o

Modules: *wilson.lop, helix.r90, compress.r90*

12.2. References

Claerbout, J., 1990, Introduction to `seplib` and SEP utility software: SEP–**70**, 413–436.

Claerbout, J., 1986, A canonical program library: SEP–**50**, 281–290.

Cole, S., and Dellinger, J., Vplot: SEP's plot language: SEP-**60**, 349–389.

Dellinger, J., 1989, Why does SEP still use Vplot?: SEP–**61**, 327–335.

Chapter 13

Entrance examination

1. (10 minutes) Given is a residual $\mathbf{r}$ where

$$\mathbf{r} = \mathbf{d}_0 - m_1\mathbf{b}_1 - m_2\mathbf{b}_2 - m_3\mathbf{b}_3$$

The data is $\mathbf{d}_0$. The fitting functions are the column vectors $\mathbf{b}_1$, $\mathbf{b}_2$, and $\mathbf{b}_3$, and the model parameters are the scalars m_1, m_2, and m_3. Suppose that m_1 and m_2 are already known. Derive a formula for finding m_3 that minimizes the residual length (squared) $\mathbf{r} \cdot \mathbf{r}$.

2. (10 minutes) Below is a subroutine written in a mysterious dialect of Fortran. Describe ALL the inputs required for this subroutine to multiply a vector times the *transpose* of a matrix.

```
# matrix multiply and its adjoint
#
subroutine matmul( adj, bb,         x,nx,  y,ny)
integer ix, iy,     adj,              nx,    ny
real                     bb(ny,nx), x(nx), y(ny)
if( adj == 0 )
        do iy= 1, ny
                y(iy) = 0.
else
        do ix= 1, nx
```

```
                x(ix) = 0.
do ix= 1, nx {
do iy= 1, ny {
        if( adj == 0 )
                        y(iy) = y(iy) + bb(iy,ix) * x(ix)
        else
                        x(ix) = x(ix) + bb(iy,ix) * y(iy)
        }}
return; end
```

Index